7-17-00

A

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE
REQUEST FOR FILING APPLICATION Under Rule 53(a),(b)&(f)
(No Filing Fee or Declaration); RULE 53(f) NO DECLARATION

PATENT
APPLICATION

Asst. Commissioner for Patents
Washington, DC 20231
BOX PATENT APPLICATION

Atty. Dkt.	72949/0269803	XP-001
	C/M#	Client Ref

Date: July 15, 2000

Sir:

1. This is a Request for filing a new utility PATENT APPLICATION entitled: **SYSTEM AND METHOD FOR COMPONENT-BASED SOFTWARE DEVELOPMENT** without a filing fee or Oath/Declaration but for which is enclosed the following:

2. **32** pages of spec, including the following:
claims (4 pages);
abstract (1 page);

3. **51** pages Appendix I (including Title page); and

4. [X] Drawings: **9** sheets (informal; 8½x11 sized paper consisting of figs. 1-12)

5.. This application is made by the following named inventors:

a. Name: **David Stanton** — Country of Citizenship: **United States of America**
Residence (City): **San Francisco, California**
Post Office Address: **c/o 44 Montgomery Street, Suite 3200**
San Francisco, CA 94104

b. Name: **Mark Potts** — Country of Citizenship: **United Kingdom**
Residence (City): **San Francisco, California**
Post Office Address: **c/o 44 Montgomery Street, Suite 3200**
San Francisco, CA 94104

c. Name: **Sameer Vaidya** — Country of Citizenship: **United States of America**
Residence (City): **San Francisco, California**
Post Office Address: **c/o 44 Montgomery Street, Suite 3200**
San Francisco, CA 94104

d. Name: **Mark Pereira** — Country of Citizenship: **United States of America**
Residence (City): **San Francisco, California**
Post Office Address: **c/o 44 Montgomery Street, Suite 3200**
San Francisco, CA 94104

PILLSBURY MADISON & SUTRO LLP

By: **Roger S. Joyner, Reg. No. 36,176**

Roger S. Joyner

1100 New York Avenue, N.W.
Ninth Floor, East Tower
Washington, D.C. 20005-3918
Tel: (650) 233-4552
Atty/Sec: RSJ/jw
Fax: (650) 233-4545

Express Mail Label: *EL 513850764 US*
Date of Deposit: *July 15, 2000*

I certify that this paper and listed enclosures are being deposited with the U.S. Post Office "Express Mail Post Office to Addressee" under 35 CFR 1.10 on the above date, addressed to Asst. Commissioner for Patents, Box Patent Application, Washington, D.C. 20231

Roger S. Joyner

Roger S. Joyner

60173192v1

SYSTEM AND METHOD FOR COMPONENT-BASED SOFTWARE DEVELOPMENT

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention is directed to computer software development and communications between computer systems. More specifically, the present invention is directed to enterprise component-based software development and to platform and architecture independent communications between disparate computer systems based thereon.

2. Background of the Related Art

The Internet has opened up the world as a single marketplace where organizations of all sizes can do business and compete. Organizations across the globe are racing to capitalize on the opportunities and increase their competitive advantage using this emergent technology. Recently, enterprise computing has been faced with ever-growing challenges now that many businesses are entering the e-commerce arena and a network economy in which transactions and information are exchanged not just inside their own enterprise, but also between enterprises in business-to-business (B2B) transactions. The architectures once defined and distinct to the enterprise and between defined enterprises are now extended to a higher level. Applications and the business functionality they encapsulate are now being offered as enterprise components within inter-enterprise business processes that connect suppliers, customers, and partners through the Internet.

The evolution of extended enterprises and virtual enterprises precludes a reliance on homogenous environments and proprietary Application Program Interfaces (APIs). Companies

have to absorb an array of differing hardware and software solutions while maintaining open integration avenues. B2B applications require infrastructure that is capable of transacting against a diverse set of software and hardware and through different integration technologies.

Nearly all enterprise systems today are transactional in nature. That is, enterprise systems define functionality in terms of sets of operations, where all the operations need to succeed or fail together as a concise unit of work. Enterprise components need to be de-coupled from any specific context in which they are used, including the transactional context, but also need to maintain the ability to enroll in transactions that may start and end outside their boundaries.

Enterprises have made use of transaction processing (TP) monitors for inter-enterprise transactional integration, but their reliance on proprietary communication protocols make managing transactions that span multiple enterprises difficult at best. Managing transactions for the extended enterprise or B2B using the Web becomes an even more daunting task.

Enterprise components in most cases encapsulate or grant access to company-sensitive information and as such need to be able to authenticate the identity of users, control user access to particular services, and provide irrefutable evidence of their involvement in a transaction (non-repudiation). Collaboration between enterprise components requires security measures that support public-key security infrastructure, such as SSL, and integration to existing enterprise security infrastructure to ensure seamless, end-to-end security. Firewalls have provided a way for enterprises to protect their information and systems, but for the extended enterprise and virtual enterprise, firewalls get in the way.

What is needed is a component-oriented framework analogous to the application framework. The framework needs to offer the flexibility of defining a standard application development model which is agnostic to the current set of middleware component models, while

structuring such flexibility in ways that deliver a consistent development model and roadmap for the development of business applications that combine components and services from the disparate environments.

SUMMARY OF THE INVENTION

In view of the above shortcomings of the prior art, it is an object of the present invention to provide an extensible platform for the construction, management and execution of component-based software.

It is a further object of the present invention to provide a component-based software development system which is largely vendor and platform neutral.

It is a further object of the present invention to provide a component-based software system capable of creating useful macro-components which provide a superior foundation for software development and reuse.

It is a further object of the present invention to provide a messaging platform capable of integrating systems and components of a heterogeneous nature and connecting systems within and across an extended enterprise.

It is a further object of the present invention to provide a component-based development system which decouples components from specific contexts in which they are used while maintaining the ability to enroll into transactions that may start and end outside their boundaries.

The above objects are achieved according to an aspect of the present invention by providing an enterprise component-based software development system capable of developing a messaging platform for communicating between computers. The development system includes a

component platform with a number of development tools and services that enable rapid and straightforward development of component-based systems. To facilitate the above functionality, the preferred embodiment of the present invention provides a component platform encompassing an open set of technologies and tools working in unison to realize enterprise component based development. The component platform defines the services and facilities as well as the structure in which components can execute as well as provides an extensible platform for the construction, deployment, management, execution and evolution of component-based software.

Further, when a component requests a service from another component, the request is serialized and encoded into an Extensible Markup Language (XML) format. The XML-encoded message is transmitted over the Internet using Hypertext Transport Protocol (HTTP) or any other transport protocol chosen, to a receiving computer, which validates the message and delivers it to the component providing the requested service. Since XML is a platform and architecture independent language, requests processed in this way can be used by a wide variety of different systems.

As noted above, one of the messaging platform's benefits is that it provides a way for disparate systems to communicate. This is accomplished through the use of supported and tested standards such as XML, HTTP, and SOAP, TIP and SSL. The messaging platform provides numerous other benefits including:

-- Choice of transport and encoding mechanisms. By default, the messaging platform uses some specific protocols, such as HTTP for message transport and SOAP for message encoding. However, users can adopt other protocols easily. For example, FTP could be used instead of HTTP and known messaging standards such as COINS or XML-RPC could be used instead of SOAP.

-- Flexible XML layout options. The messaging platform allows users to determine the layout of their XML documents to structure them so that they will be understandable to remote systems.

-- Choice of DOM implementation. As is known in the art, a Document Object Model (DOM) determines how the information in an XML document is read and processed by the system. Different companies and organizations have created numerous DOM implementations. A messaging platform in the system preferably uses Sun Microsystems' DOM by default; however, one can use any other DOM instead. In fact, the messaging platform provides users with the flexibility to implement any DOM and SAX combination. For example, one could implement Sun's DOM and IBM's SAX parser.

-- Choice of schema. As is known in the art, schemas are documents that describe the information contained in XML documents. The messaging platform provides users with some default schema, but does not limit them to their use. The messaging platform supports any chosen schema. This means that one can use a standard schema, a standard schema with some customization, or a schema that is unique.

-- Schema generation. The messaging platform provides a GUI tool called the schema generator that automates the creation of schemas and the serialization of objects. The schema generator's GUI provides a simple and quick way to select the fields to be included in the schema and its serialized object, thus making it easy to modify schema and serialized objects during the development process.

-- Object neutrality. The messaging platform accommodates for differences in the way systems "think about" information. For example, a sender system might conceptualize an entity that contains customer information as "Customer." However, the recipient system might

consider that information to be and entity that it recognizes as a "Supplier." Despite their differences, these systems can exchange customer information because the messaging platform transmits data in a neutral manner through the use of schemas. As long as the two systems agree to use the same schema, they can exchange information and collaborate seamlessly.

-- Support for XSL. Because not everyone will want to encode objects using SOAP, the messaging platform supports the use of XSL for formatting and presenting complex XML documents. The messaging platform provides SAXON 4.5 as the XSL engine; however, a user can replace it with the XSL processor of his choice.

Component-based Development (CBD) today is recognized as the best approach to reducing costs and time to market for the development of business systems. CBD has been defined as an approach in which all artifacts, including executable code, interfaces, architectures, of disparate granularity can be built by assembly, adapting and 'wiring' together existing components into a variety of configurations.

Enterprise Component Platform (ECP) is a standards based comprehensive solution to developing enterprise-scale distributed systems, through a holistic component based approach. ECP provides frameworks, prefabricated enterprise components, and an architecture and toolkit that support the definition, development, assembly, deployment, execution, management and evolution of enterprise components.

By defining systems in terms of federations of enterprise components, ECP addresses the challenges of controlling the complexity and risk in developing large, enterprise-level distributed systems in a way that supports the iterative, incremental development, management and evolution of systems.

ECP defines components as any artifact that is specified, constructed and deployed as a discrete unit. The ECP architecture and tool set provides the same services and facilities to components of all levels. However, the ECP focus is on enterprise components needed for enterprise-scale, distributed component-based systems.

Enterprise components systems as well as units of aggregation ECs can also be integrated and assembled into federations that extend the enterprise (supporting the extended and virtual enterprise models). By supporting the native protocols, as well as Internet protocols (such as HTTP, HTTPS, SSL, SOAP, and TIP) and semantic data standards (common repositories of XML-schema that can be shared between enterprises and their systems), enterprise components can integrate and collaborate, in a flexible and evolutionary manner, with systems outside their direct architectural scope. For example, enterprise components can access the services of a different remote enterprise, even when the services are offered from a proprietary architecture/technology, such as an ERP system.

Enterprise components are autonomous (outside their declared dependencies on other defined components) but still require an execution environment to be deployed to. The enterprise component platform leverages and augments the commercially available execution environments for distributed components (e.g., EJB: iPlanet iAS 6.0, BEA WebLogic 5.1, IBM WebSphere 3.0). The ECEE augments these execution environments offering features not supported by individual models but required for enterprise component-based development.

ECP also supports integration between enterprise components developed and deployed with disparate distributed component technologies and execution environments as well as other middleware such as message oriented middleware (MOM) for asynchronous messaging.

ECP is the only comprehensive solution for controlling the complexity and risk in developing large-scale distributed systems, through a holistic component based approach that:

-- Defines components at the next architectural level, above distributed components, allowing for the development of components from disparate component models, developed in different languages, within a heterogeneous environment.

-- Supports and augments the leading component execution environments, allowing them to be integrated into a single enterprise component execution environment (ECEE) to deploy enterprise components.

-- Eases development complexity by supporting a discrete recursive breakdown of problem domains in terms of components, that is architecture and reuse centric.

-- Reduces time to market and cost by isolating the development, deployment and maintenance of enterprise components. ECP not only supports high levels of concurrent development but also leverages internal skills and a cost effective, consumer-producer model for iterative incremental system development.

-- Provides component frameworks and templates for the development of additional components to further reduce time to market and costs of development, and reduce the learning curve and burden on developers and architects.

-- Provides a logical component repository and component transporter that to enable easy sharing of components, installation management, and incremental upgrades over time.

-- Covers the entire lifecycle of enterprise components from design through definition, development, assembly, execution, to management.

-- Enables efficient leverage of existing solutions, prefabricated components and architectural artifacts when developing new custom enterprise components.

-- Allows for the efficient evolution of installed solutions through an inherently flexible architecture, supporting adaptability, integration and collaboration through open standards and technologies.

BRIEF DESCRIPTION OF THE DRAWINGS

These and other objects, features and advantages of the present invention are better understood by reading the following detailed description of the preferred embodiment, taken in conjunction with the accompanying drawings, in which:

FIG. 1 is a system view of two computer systems communicating according to a preferred embodiment of the present invention;

FIG. 2 is a diagram of component layers in the preferred embodiment;

FIG. 3 is a block diagram of elements of a component platform according to the preferred embodiment;

FIG. 4 is diagram of a component hierarchy in the preferred embodiment;

FIG. 5 is a diagram of a lookup process according to the preferred embodiment;

FIG. 6 shows granularity of structures in the preferred embodiment;

FIG. 7 is a diagram of elements and component layers of an enterprise component platform according to the preferred embodiment;

FIG. 8 shows granularity of structures in the preferred embodiment;

FIG. 9 is a block diagram of elements of a component platform according to the preferred embodiment;

FIG. 10 is a block diagram of elements in a messaging platform according to the preferred embodiment;

FIG. 11 is a diagram of the component repository; and

FIG. 12 is a diagram of the transport between component layers in the preferred embodiment.

DETAILED DESCRIPTION OF PRESENTLY PREFERRED EXEMPLARY EMBODIMENTS

An overall diagram of a preferred embodiment of the present invention is shown in FIG. 1. On perhaps a topmost level, the system includes a first computer system 10 which can communicate with a second computer system 15 over a communication network 20. The network 20 may be a dedicated one between the two systems. It may include other computer systems which are not involved in communications between the first and second computer systems and are not shown in the Figure for simplicity and ease of explanation. The network 20 may be an external network such as a distributed communication network such as the Internet. The first and second computer systems 10 and 15 may interface directly with the network 20, or they may be connected to the network via proxies, servers or the like. All of these variations are deemed to be within the scope of the present invention.

An enterprise component may be thought of as any piece of software that is specified, constructed and deployed as a discrete unit. It is a unit of composition for larger scale components and may itself comprise other components. A component offers a set of services to its consumers and itself may rely on another set of services. It may be distributed in nature and may execute over one or more processes in a heterogeneous computing environment. The messaging platform enables the sharing of enterprise component services existing within the enterprise as well as those existing in other, networked enterprise systems.

To facilitate the above functionality, the preferred embodiment of the present invention provides a component platform 105 (see FIGS. 2, 9 and 12) encompassing an open set of technologies and tools 110-135 working in unison to realize enterprise component-based development. The component platform 105 is an architecture for the development and management of enterprise-scale multi-tier, multi-user applications 100. It defines the services and facilities as well as the structure in which components can execute. Along with the processor layer 145 and custom layer 140, it provides an extensible platform for the construction, management, deployment, execution, evolution and execution of component-based software.

Component platform tools 110-135 are completely open and easily extensible. The elements of the component platform 105 are preferably written in Java or a similar language to make them largely platform independent to maximize their utility. Metadata required by the tools 110-135 is stored in an XML file called the component repository within the file system in an organized fashion to allow developers to integrate other tools at will. Component metadata is implementation- and application- specific; however, it will generally include the following:

-- name or other identification information;

-- component category and classification, e.g., GUI, Server, Viewer, etc.;

-- dependencies (if the particular component must have access to another component in order to operate properly; and

-- version or replacement histories, deployment information and the like.

It should be noted here that the component repository is maintained by the system to reflect the most current component dependency situation. That is, assume a first component A indicates in its metadata dependency information that it is dependent on a second component B, i.e., A must have access to B in order to function properly. If the component repository then receives a third

component C which its deployment information indicates is to replace component B, component A's metadata dependency information is update to reflect that it is now dependent on component C.

Management of components is described through their dependency specification, versioning and deployment. The component platform 105 enables specification of component versions, type information, structures and relationships for a heterogeneous, distributed computing environment. These semantics are specified and associated with the component 225 (see FIG. 3) in a tool called the component assembler 240. The component assembler 240 captures this information in XML files 205 and generates cross-platform scripts and command launch items 210 that are interpreted at execution time by a component launcher 220. These scripts and launch items and convert the general component specifications from the component assembler 240 into a platform-specific runtime package. For example, given a component description from the component assembler 240, the scripts may convert the component description to a Java description or a C++ DLL description. This allows developers to build and manage macrocomponents including a variety of software elements such as JSP/servlets, HTML pages, EJBs, JavaBeans and supporting Java framework classes. Additionally, OS executables can be developed within the same environment so that non-Java components can be managed as well.

Every system participating in component execution must include a component repository 230 (shown in FIG. 11) which physically hosts all its components 225. The component repository 230 has two sections, a component data section where the components 225 are stored, and a shared data section which holds data to be shared between components as well as data that may be dynamically changed at runtime.

All components 225 in the component platform 105 must adhere to a certain physical structure in the file system 230. This structure includes an organization of directories and

supporting metadata files. The tools in the component platform 105 logically look at this structure as a component repository 230. This approach enables the component platform tools 110-135, as well as development tools such as IDEs, versioning tools and testing tools, to work with components 235 in an open, non-intrusive manner.

Component supplements, including source code, DLL/DML scripts, documentation, online help, design models and test scripts can be managed together with the executable binaries for the component 235 and made available where appropriate.

Completed components 225 can be distributed to the component repository for developers to assemble larger components or for applications for upgrades via the component transporter 235. The component transporter 235 serves as a catalog of components 235 made available to appropriate users. This may be done when a component 225 is executing and the component transporter 235 obtains a component having certain characteristics which the executing component indicates it needs, or it may be done when a new component is installed to determine what other components are available to it. Access may be limited to particular users with a user authentication and restriction scheme as is known in the art.

When components 225 are installed from the component transporter 235 into the component repository 230, install scripts are executed to set up additional information for the component 225 in metadata. Similarly, uninstall scripts are executed to clean up behind the component 225 when it is removed from the component repository 230. New components declared to be backward compatible with existing components 225 in the repository can automatically replace the older version upon installation of the new version.

The component assembler tool 240 gives developers a way to create new types of components 225 by offering a number of predefined templates and wizards 245. These can be used

to get a starting code line that includes all necessary relationships and versioning information pre-populated. Further, custom templates can be created to extend the assembler. Further development of the components 225 may be done by software development tools as is known in the art.

Java Remote Method Invocation (RMI)-based distributed components 345 can be built based on the component runtime 135, and the component builder tool can be used to specify remote interfaces and provide an interface for writing remote callable Java objects. The builder generates code complying with the component runtime 135; for example, it might include Java source codes. This is in contract to the component assembler 240, which attempts to generate components 225 in a platform and deployment environment agnostic format, using metadata and the like.

A large part of many project developments is spent in hand coding object-to-schema mapping. The component platform's constructor tool is a tool for mapping component data to relational database or XML schema. It is used to describe the mapping relationships between business objects (lower level object which may be used to develop components) and relational tables, or standard XML schema definitions. Inheritance, association and aggregation techniques between business objects are supported.

As shown in FIGs. 6 and 8, enterprise components are larger in terms of granularity than distributed components, but are always constructed of other components. For example several individual distributed components might be combined to provide a more meaningful business service that would be classified as an enterprise component. Enterprise components are capable of spanning multiple tiers of architecture (user services, web services, business services, and data services), and offer the ability to partition and deploy in a flexible manner, executing over one or more processes in a heterogeneous environment. While enterprise components themselves don't

dictate a physical environment or architecture, their more granular constituents may dictate its deployment configuration into a physical architecture/environment. Enterprise components themselves are units of composition meaning their constituents can be other enterprise components. This allows for component to be developed as white-box or black-box components at differing levels of granularity.

Enterprise components can themselves be a unit of composition for larger scale components, and/or can be associated to form federations of components that support major aspects of an enterprise's business.

The component runtime component 135 (see also FIG. 7) manages execution of distributed components across supported runtimes such as component runtime, WebLogic runtime by BEA Systems, Incorporated of San Jose, California USA, and OS runtime. Features such as version management are made possible via the component runtime 135. Thus, the runtime component 135 can handle components integrated at different levels for different runtime models. These can then be transparently integrated into runtime packages based on standard component models such as J2EE or COM+. This may not be necessary for the component runtime package if it is able to understand the components without integration.

Frameworks, through the implementation and aggregation of patterns, answer both the limitations of class libraries in supplying an architectural infrastructure and reusable design. The architectural infrastructure of a framework dramatically decreases the amount of lower level plumbing code that the developer has to implement and understand. It defines the overall structure of an application, the partitioning into classes and objects, the key responsibilities thereof, how the objects collaborate, and most importantly accepts the responsibility for the control and flow of execution. By taking over the responsibility for the execution control of an application, and

removing this responsibility from the developer and the code he has to implement, the developer can focus his efforts on the particular business problem at hand.

A framework should emphasize design reuse over code reuse. A framework should reduce design decisions. The use of frameworks means the reuse of a main body of code and development of the functionality it calls. Developers therefore must code within specified interfaces defined by the frameworks, which reduces the number of design decisions they have to make. Also, a framework should standardize the application structure. Through the defined interface that frameworks use to call back to application-specific functionality, applications built using frameworks have similar structures and are therefore more consistent and easier to maintain. A framework also should be extendable by domain designers. Customization is achieved through sub-classing abstract classes or implementing interfaces from the individual frameworks.

The business component framework 330 provides the infrastructure to build transactional business components consistently across a variety of execution environments and transactional resources, including servlet engines 340, EJB containers and component runtime 135. Due to its modular, layered implementation the business component framework 330 enables incremental construction of complex server behavior encompassing caching 345, concurrency, security 350 and the like. Complex relationships of business objects, including inheritance, association and aggregation can be transacted consistently while maintaining atomicity, consistency, identity and durability (ACID) properties in the object world. Persistence implementation is flexible and enables advanced performance enhancements like lazy instantiation for incremental retrieval.

Business collections enable business transactions to be performed on a collection of business objects instead of on each individual business object. This considerably simplifies coding

and improves performance, as each operation on a collection can be a single transaction. The state of each object in the collection is maintained so that when a series of changes to the collection is saved, only the necessary operations are performed on the objects in the collection. Typical collection operations like sort and search are well-defined by following standard patterns consistent with the Java collections framework.

Another alternative is the high transaction batch processing framework 380. It provides the architectural foundation to building high speed, high volume, file-to-file, file-to-dB and dB-to-file batch jobs in Java. This framework includes a COBOL-based API syntax for extremely flexible manipulation of data structures; buffered file reads and writes for very high operational speeds; and targeting to high speed and ease of use in an object-oriented programming environment.

The preferred embodiment of the present invention preferably additionally includes a logging component 360 which provides an API to write log messages to the shared data area of the component repository that can be filtered by an application-defined criteria and criticality at execution time. The events that are logged are application-specific and user-chosen; typically, events such as user logins and logouts, error exception conditions and the like might be included in the events logged. Messages from the logging component 360 can be written to any user-defined output stream. XML output formatting of the XML-formatted logging data can be turned on to provide filtered viewing. Log messages from various sources can be consolidated to track distributed transactions. That is, logging components 360 in a number of enterprise systems could be configured to report logged events to a single shared data repository, from which the data could be consolidated to follow the progress of a distributed process through the enterprise systems.

Additionally, a caching component 345 provides a service to cache any type of Java objects in a distributed environment. The caching mechanism is delivered through the creation of application-defined cache managers within a Java Virtual Machine (JVM) that coordinates cached objects. The caching component includes functionality for multiple key cache access; partitioning based on class; Least Recently Used (LRU) memory management; singleton RMI read/write cache; multi-JVM remote administration; and administration tools and statistics.

A security component 350 provides a complete set of classes for implementing robust security architectures, extending Java security models to provide a consistent set of security APIs for all application elements. Along with a security administration component 365, the security component 350 simplifies the implementation of security for a three-tier architecture having a user services tier, a business services tier and a data services tier, and automatically and seamlessly handles simple security constraints while its robust Access Control List (ACL)-based architecture ensures that the most sophisticated security requirements can be properly encapsulated. The security component includes authorization services available to servlets 340, EJBs and other application objects; a servlet authentication service; and generic security constructs based on ACL, permissions and roles.

A workflow component 375 offers integrated workflow capabilities that enable the separation of application logic from process logic. The workflow component 375 introduces the concept of a business process object, which represents a set of one or more linked tasks. Each task created within a business process is represented by a business activity object. The vendor-specific session context is contained within a generic workflow source manager to isolate the developer from vendor-specific APIs and permit the workflow objects to be mobile. The workflow component 375 includes business activity and business process strategizable objects; a workflow

session manager; independence from vendor-specific workflow engine APIs; and support for the Verve process engine from Verve, Incorporated of San Francisco, California USA.

Core technical components 315 provide a solid technical foundation for building robust, high performance, secure, complex, large-scale components. Core components 315 offer technical services common to all applications 140. They are comprehensive in their applicability and customization is not required.

Horizontal/integration components 320 integrate diverse execution models consistently. Horizontal/integration components define the interface and protocol between disparate execution models. Like the core technical components 315, they are comprehensive and customizations are not required. Horizontal/integration components 320 may be used in workflow 375, e.g., Verve and Conductor (made by Epicor Software Corporation of Irvine, California USA and Forté Software of Oakland, California USA) architectures; browser interfaces, e.g., JSPs and servlets; packages such as Ariba (From Ariba Corporation of Mountain View, California USA), SAP and Peoplesoft (Pleasanton, California USA); MOM integration, e.g., MQ and MSMQ technologies; communication mechanisms such as e-mail and fax; dynamic pricing and auction modules, e.g., Moai (San Francisco, California USA); content management such as Open Market (Burlington, Massachusetts USA); and personalization such as NetPerceptions (Eden Prairie, Minnesota USA).

A vertical component 325 defines a specific business problem which it addresses. It may or may not need customization. Vertical components may be non-industry specific, e.g., e-commerce, knowledge management, bill presentment, and customer relationship management; or industry-specific, e.g., telecommunications, financial services, healthcare, retail, manufacturing, entertainment, and procurement. For example, vertical components in a procurement application

may include components addressing specific aspects of shipping, tax, orders, party/organization/person, business role, fulfillment plan, inventory items, role profiles, and contact relationships.

Component invocation integration is the sharing of component services that may exist within an enterprise or outside the enterprise. By allowing access to services provided by components contained inside or outside any given application, organizations are able to share both data and processes among many logistical applications and therefore integrate those logical applications into larger applications. While component invocation integration is the most flexible and desirable level of integration, it is also the most difficult to achieve. For component integration to be feasible there needs to be a common protocol for component communication.

The present invention solves the above problem by providing a messaging platform for communication between components. A messaging platform 25 is shown in FIG. 10 and is formed by parts of the first and second computer systems 10 and 15 and the network 20. The messaging platform 25 is a middle layer in a multi-tier communication architecture such as the one shown in FIG. 2 and facilitates communication between system components such as the first and second computers 10 and 15. The messaging platform preferably has three main parts: a request processor 30 which receives a request 50 for a component service (possibly in a format not understandable by any component providing that service) from a component 225 operating on the first computer system 10, validates it, converts it to a message 35 having a platform- and runtime-independent representation, and passes it to the network 20; a transport mechanism, i.e., the network 20, for conveying the message from the first computer 10 to the second computer 15; and a request processor 40 in the second computer 15 for receiving the message, converting it to a form

55 usable by the second computer system 15 and providing it to a service component 45 of the second computer system 15 which can provide the requested service.

During initialization of the service routine 45 or at another appropriate time, as shown in FIG. 5 the service routine 45 registers with its request processor 40 to notify the processor 40 of the services offered to other system components by the service component 145 in Step 405 for validation purposes. This is done by providing the request processor 40 with the service routine's entry points, i.e., calls to invoke the advertised functionality, as well as other information to distinguish the component 45 from other service components 45 providing the same services. The service component 45 also provides the request processor 40 with various parameters including a description of the advertised functionality through a distributed component interface. The request processor 40 saves the parameters and their structure so that it can validate requests that are subsequently made of the service component 45.

When a component in the first computer system 10 wants to ask that a service component to provide some service, it sends a component invocation request to that effect to its request processor 30 in Step 410. This is a native language message which is intercepted by the request processor 30 and converted to XML or another common format in Step 415. In Step 415 the lookup service 430 finds a component that is able to complete the requested service. The component selection can be refined based on other criteria accompanying the request and known attributes of the component as described above. Additionally, the selection may be made on real-time dynamic information relating to each service component 45, e.g., queue depth, average compute time and network latency. Based on this, the lookup service 430 sends the XML message to the identified component 45 in Step 435. The XML message is then received and processed by the receiving system.

The component invocation request 410 and any associated parameters are serialized by the messaging platform 25 in Step 425 so that they may be converted to a platform- and runtime-independent format 35 suitable for transmission over the network 20. In the preferred embodiment, the transmission format 35 is the Extended Markup Language (XML) and an example request is shown in FIG. 5. As is known in the art, XML is a subset of the Standard Generalized Markup Language (SGML) designed especially for web documents. It allows system architects to create customized tags which enable the definition, transmission, validation and interpretation of data between different entities. Once the component invocation request has been coded in XML, it can be accessed and used by any computer system capable of receiving the request and converting it to its own internal format for further processing.

Translation of a service request 35 into XML may be done by an automatic software gateway on the first computer 10 side of the messaging platform 25 or by a separate encoding processor 60 as described below. To guarantee the uniform interpretation of the service request 35 and any accompanying parameters a generic XML vocabulary is necessary. The request can then be parsed and understood by the remote component gateway.

Preferably, to accomplish this task the encoding processor 60 in the messaging platform 25 converts the XML documents 35 into transmissible objects 65 using the Simple Object Access Protocol (SOAP), which uses the Hypertext Transfer Protocol (HTTP) as the transport mechanism and the serialized XML document 35 as the payload. SOAP provides a way for applications to communicate over the Internet independent of their platforms. SOAP specifies how to encode an HTTP header and an XML file so that a program in one computer can call a program in another computer using an appropriate transmission medium, pass it information, and possibly

receive a reply from the recipient. In the preferred embodiment, the transmission medium is the Internet which is used as network 20. A sample SOAP-encoded request might be

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

<SOAP-ENV:Envelope
     xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
     SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
     <SOAP-ENV:Body>
          <m:GetLastTradePrice xmlns:m="Some-URI">
               <symbol>DIS</symbol>
          </m:GetLastTradePrice>
     </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Another benefit provided by the use of SOAP should not go unmentioned. The use of SOAP allows system communications to generally pass through system firewalls and thus provides expanded usability. Most distributed object protocols suffer from firewall blockages because they transmit over ports which are blocked by the firewalls. However, most firewalls pass traffic on port 80, the channel on which most HTTP communications, such as those generated by SOAP, are conducted. Thus, HTTP-encoded transmissions such as the above can easily pass through.

It should be noted that as an alternative to HTTP transport, the invention may employ HTTP authentication mechanisms as well as SSL for secure channel communications (using HTTPS).

The preferred embodiment also includes a translation component (not shown) which uses an Extensible Stylesheet Language (XSL) that allows users to implement SOAP for encoding objects. For systems that do not use SOAP, XSL provides a way to transmit information in the

format that the system understands. In the preferred embodiment, the XSL engine is Saxon 4.5. Other known standard XML vocabularies may be used in place of SOAP or XSL, e.g., XML_RPC or COINS (this assumes that both communicating systems speak XML_RPC, COINS, etc.). Such variations will be understood as being within the scope of the present invention.

Finally, in Step 425 the messaging platform 25 uses the HTTP protocol to transport the SOAP-encoded XML message 65 over the Internet 20 between the first and second computers 10 and 15. Address information from this purpose comes either from component metadata or data from the request processor 40. It also is affected by the type of protocol being used -- for example, connectionless protocols such as SOAP which do not require a full connection model may not need such information, while connectioned protocols such as Java RMI may need this additional information.

When the second computer 15 receives the transported XML document 65 from the network 20, its request processor 40 checks to see if the destination service component 45 identified therein is a registered component on that system. If not, or if the service component 45 is in fact registered but the request 35 does not have the correct format, number of parameters or the like, an appropriate error message is returned to the first computer 10 (again, in a SOAP-encoded XML format). The error message preferably specifies the nature of the error. If, however, the service provider 45 is properly registered and formatted, the component invocation request 35 is passed to the service component in question to be processed.

The connection assembler is used to create, manage, and manipulate messaging platform connections. These include connections from external systems to the messaging platform and from the messaging platform to external systems. For example, there are generally two relationships components can have, aggregation and run-time connection, and the connection

assembler can manage such aggregation contacts. When a Java application makes a SOAP call, for example, it expects a stub in return. If the receiving system does not speak Java, the connection assembler can take care of generating a stub to be supplied to the original Java application.

The messaging platform provides a tool called a schema generator that is used to create XML documents containing the information to be transported to another system. As is known in the art, when the syntax and semantics two systems use to communicate with each other is known in the art, an XML construct called a schema can be used to define the protocol and format messages for transmission between the systems. The schema generator also greatly simplifies the creation of the XML schema by providing a Graphic User Interface (GUI) that automates the creation of schemas and the serialization of objects. A schema is a set of predefined rules that describe a given class of XML document. A schema defines the elements that can appear within a given XML document along with the attributes that can be associated with a give element. It also defines structural information about the XML document. The schema generator's GUI provides a simple and quick way to select fields to be included in the schema and its serialized object, thus making it easy to modify schemas and serialized objects during the development process.

As an alternative to custom-made schemas using the Schema Generator above, standard schemas from organizations and standards bodies may be used, or a combination of the two may be used. Such variations will be understood to be within the scope of the present invention.

The messaging platform also includes a transport framework for the implementation of HTTP for transport. Although Messaging Platform uses the HTTP over the Internet as its default transfer protocol, open interfaces are provided to help developers implement the protocol of their

choice. This is because the transport of a request done through a framework can be specific to a particular transport, e.g., HTTP, FTP, SMTP and the like.

Nearly all enterprise systems today are transactional in nature. That is, enterprise systems define functionality in terms of sets of operations, where all the operations need to succeed or fail together as a concise unit of work. Enterprise components need to be de-coupled from any specific context in which they are used, including the transactional context, but also need to maintain the ability to enroll in transactions that may start and end outside their boundaries.

The messaging platform addresses this problem by making use of the Transaction Internet Protocol (TIP). TIP enables messaging platform to achieve atomic commitment of a two-phase commit protocol between enterprise components. TIP enables the messaging platform to coordinate transaction managers from disparate systems communicated via differing protocols using a "bring your own transaction" (BYOT) mechanism. Using the BYOT mechanism, the messaging platform attaches transactional context to messages it sends and receives allowing a controlling transaction manager to coordinate sub-transactions, controlled by remote transaction managers, that have been enrolled into the original transaction. In this way, TIP can be used to allow transaction management across distributed transactions which may be distributed across the Internet or another communications network.

To enable TIP the transaction managers that want to support the protocol must map their internal transaction identifiers into this TIP format. Each transaction manager uses its own identity mechanism for its localized transaction, and the identifiers are exchanged when the relationship between Transaction Managers is established. Transaction Managers therefore use their own format of transaction identifier internally, but hold foreign identifiers for each

subordinate transaction from distributed Transaction Managers involved in the distributed transaction.

Additional aspects of the messaging platform according to the preferred embodiment are described in the "Messaging Platform Overview" (June 2000) by Xpedior Products and Tools Group, contained in Appendix I and incorporated herein by reference.

The present invention has been described above in connection with a preferred embodiment thereof; however, this has been done for purposes of illustration only, and the invention is not so limited. Indeed, variations of the invention will be readily apparent to those skilled in the art and also fall within the scope of the invention.

WHAT IS CLAIMED IS:

1. A component platform for development, deployment, management evolution and execution of components in a component-based software development system, the platform comprising:

a component assembler for generating components and data associated therewith and for storing the components and associated data in a component repository;

a component transporter for distributing the components; and

a component launcher for executing the components responsive to the associated data; and

a component browser for managing components and their associated data stored within a component repository.

2. The platform of claim 1, wherein the associated data is stored in the component repository according to a predetermined structure accessible by external software components.

3. The platform of claim 1, wherein the associated data is stored in a platform and application-independent format.

4. The platform of claim 1, wherein the associated data includes at least one of component versions, type information, structural descriptions and dependency relationships.

5. The platform of claim 1, wherein the component assembler is for executing install scripts when storing the components in the component repository.

6. The platform of claim 1, wherein the component assembler is for executing uninstall scripts when removing the component from the component repository.

7. The platform of claim 1, wherein the component assembler generates the components based on at least one of a template and a wizard.

8. The platform of claim 1, further comprising a component runtime for managing execution of the components.

9. The platform of claim 1, further comprising a logging component for writing log messages from components and their centralized collection.

10. The platform of claim 1, further comprising a caching component for caching at least one of components and any other identifiable artifact.

11. The platform of claim 1, further comprising a workflow component for separating application logic from process logic within the platform.

12. The platform of claim 1, further comprising a security component for managing authentication and authorization within the platform.

13. A messaging platform for a component-based software system, the platform comprising:

a connection assembler for at least one of creating, managing and manipulating a first messaging platform connection;

a protocol management framework for implementation of a predetermined transport protocol over the first connection;

a schema generator for, responsive to a request for service received over a second messaging platform connection, creating a document according to a predetermined format, the document containing information to be provided to another system over the first connection;

an encoding component for converting a document in the predetermined format into a first encoded object that can be understood and used by the another system, the first encoded object being encoded according to a default encoding protocol; and

a translation component for encoding a document in the predetermined format into a second encoded object that can be understood and used by the another system, the second encoded object being encoded according to an encoding protocol different from the default encoding protocol.

14. The platform of claim 13, wherein:

the information is provided by a service component; and

the request for service is in a form not understandable by the service component.

15. The platform of claim 13, wherein the service request is in a platform and application-independent format.

16. The platform of claim 15, wherein the service request is in an Extensible Markup Language format.

17. The platform of claim 13, further comprising a lookup service for determining a service component to handle the service request.

18. The platform of claim 17, wherein the lookup service determines the service component to handle the service request based on information associated with the service component.

19. The platform of claim 13, wherein the protocol management framework implements HTTP for transport.

20. The platform of claim 13, wherein the default protocol is SOAP.

ABSTRACT OF THE DISCLOSURE

An enterprise component-based software development system includes a component platform with a number of development tools and services that enable rapid and straightforward development of component-based systems. The component platform describes a middle tier architecture for a multi-tier, multi-user application. It defines the services and facilities as well as the structure in which components can execute. It helps to provide an extensible platform for the construction, management and execution of component-based software. A messaging platform facilitates communication between different computers. When a component requests a service from another component, the request is serialized and encoded into a platform-independent language such as XML. The XML-encoded message is transmitted over the Internet using an HTTP protocol to a receiving computer, which validates the message and delivers it to the component providing the requested service. Since XML is a platform and architecture independent language, requests processed in this way can be used by a wide variety of disparate systems.

APPENDIX I

xpedior™

Products and Tools Group

Messaging Platform Overview

June 2000

Xpedior, Inc.
San Francisco, Los Angeles, Denver,
Boston, London(UK), Perth(Australia)
www.xpedior.com

Table of Contents

Preface

Audience and Purpose

Messaging Platform Overview is an introduction to Messaging Platform for both technical and non-technical readers. It provides both high-level and detailed descriptions of what Messaging Platform does and how it works. It also explains the benefits Messaging Platform provides and describes the tools, components, and frameworks that make up Messaging Platform. It also provides information about how to create a Messaging Platform component using the Assembler Component and how to use Messaging Platform monitoring tools.

Document Overview

Messaging Platform Overview contains five sections:

Introduction to Messaging Platform. This section explains what Messaging Platform is and how it works at a high level. It provides scenarios illustrating Messaging Platform's use. It also describes the tools, components, and frameworks that comprise Messaging Platform and discusses the benefits provided by Messaging Platform.

Inbound Message Process. This section describes the role of Messaging Platform in receiving requests from and external sender, processing the requests for use by the recipient, and returning a response from the recipient to the original sender.

Outbound Message Process. This section describes how Messaging Platform sends requests from a Component Platform component to an external service.

Messaging Platform Component Templates. This section describes the Messaging Platform templates that can be used to create Messaging Platform components.

Monitoring Tools. This section describes Messaging Platform monitoring tools.

Additional Information

This section contains URLs to Websites that provide additional information about some of the topics discussed in this document.

HTTP References

The Hypertext Transfer Protocol (RFC2068: Hypertext Transfer Protocol) is available at: http://info.internet.isi.edu/in-notes/rfc/files/rfc2068.txt

An addendum to the protocol is available at: http://www.w3.org/Protocols/History.html

XML References

The XML Specification is available at: http://www.w3.org/TR/WD-xml-lang.html

The XML-Data Specification is available at: http://www.w3.org/TR/1998/NOTE-XML-data/

Document Content Description for XML is available at: http://www.w3.org/TR/NOTE-dcd.html

Information about Namespaces in XML is available at: http://www.w3.org/TR/REC-xml-names/

Information about the XML Linking Language is available at: http://www.w3.org/1999/07/WD-xlink-19990726.html

A description of XML Schema Part 1: Structures is available at: http://www.w3.org/TR/1999/WD-xmlschema-1-19991105/

A description of XML Schema Part 2: Datatypes is available at: http://www.w3.org/TR/1999/WD-xmlschema-2-19991105/

SOAP References

The Simple Object Access Protocol (SOAP) specification is available at: http://msdn.microsoft.com/xml/general/soap_v09.asp

Introduction to Messaging Platform

This section explains what Messaging Platform is and how it works; provide some scenarios that illustrate its use; describe the tools, components, and interfaces available for it; and describe its features.

What Is Messaging Platform

Messaging Platform provides a mechanism for connecting heterogeneous systems within and across enterprises. Using Messaging Platform, you can make your enterprise services available over the Web. Your system can also use Messaging Platform to access the services of a different remote enterprise. Messaging Platform can be implemented within your enterprise, via an intranet, to provide communication between disparate systems. Finally, Messaging Platform provides the flexibility to work with proprietary systems, such as an ERP, through the use of adapters and connectors.

Messaging Platform uses standard Internet protocols to make cross-enterprise exchanges possible. Messaging Platform converts information from application-specific formats into XML documents, transports the documents between senders and recipients via a protocol (such as HTTP), repackages the XML documents for use by the recipient, and returns information from the recipient back to the sender. Messaging Platform implements Microsoft's Simple Object Access Protocol specification (SOAP), which describes a way to use HTTP as the base transport mechanism for exchanging XML documents. SOAP is an emerging Internet standard.

Note: Although SOAP the Messaging Platform's default transport mechanism, you can implement other mechanism in its place. Messaging Platform has been designed to allow flexibility in the adoption of standards and protocols.

How Does Messaging Platform Send and Receive Information?

The XML documents exchanged via Messaging Platform represent objects or, more precisely, contain the methods and parameters needed to invoke a function that can return specific information to a sender. In effect, Messaging Platform provides a way for an application to make its existing methods available as services that other applications can use. Likewise, it provides a way for an application to access the services offered by a remote application. Thus, with Messaging Platform, your existing enterprise is transformed into a Web-enabled enterprise.

At a high level, Messaging Platform performs three functions:

Generates and encodes XML documents from serialized objects. When an object is serialized, it is converted into a stream of bytes. Messaging Platform serializes objects and then packages them as XML documents that conform to the specific requirements of SOAP or a different encoding mechanism of your choice.

Transports XML documents. Messaging Platform uses HTTP to transport XML documents between a sender and a recipient. To ensure that XML documents are sent and received correctly, Messaging Platform establishes and stores connection information. When it receives an XML document from a sender, Messaging Platform makes sure that the intended recipient exists, and it then makes sure that the message is transmitted to the recipient correctly. Although HTTP is Messaging Platform's default transport protocol, other protocols can be used in its place.

Runs requested services. The XML documents created by Messaging Platform contain three types of information: the name of the service; the method to be run by that service; and the information, or parameters, to be given to these services. Messaging Platform takes these parameters from the XML documents and passes them to the requested service. Messaging Platform subsequently sends the information returned as a result of running the service to the requestor.

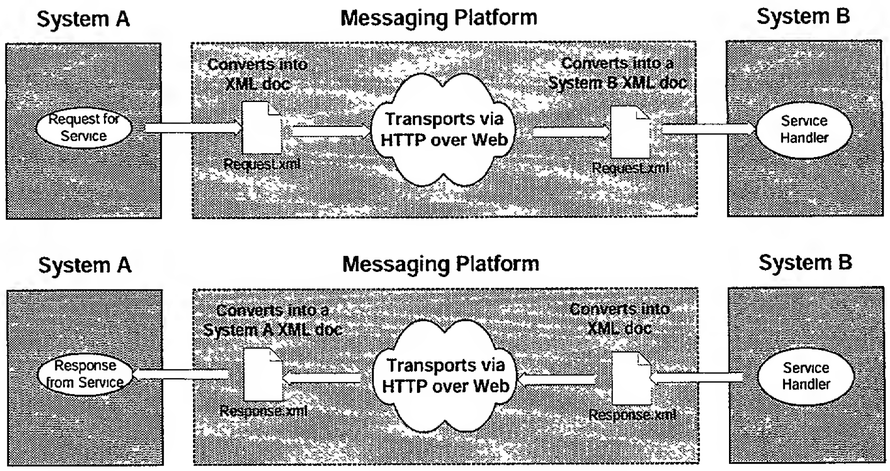


Figure 1. High-level illustration of Messaging Platform

The secret to Messaging Platform lies in its use of XML schemas. A schema is a document that presents a collection of rules for naming and defining objects, their properties, and their attributes. Two systems can use Messaging Platform to exchange object data packaged in XML documents if they use the same schema. As long as the XML documents exchanged between the systems are formatted in compliance with the schema's rules, the systems will be able to invoke each others' methods.

Messaging Platform Scenarios

This section describes some scenarios that illustrate how Messaging Platform can be implemented across or within enterprises. For these scenarios, we've invented a .com company called Car.com. Car.com provides services for car buyers. For example, buyers can purchase or lease a vehicle over the Web, arrange for its financing, and shop for and purchase for car insurance. To provide these interactive services, Car.com's system must be able to exchange information in real-time with the systems of many different companies. Messaging Platform can make this possible.

To illustrate some examples of Messaging Platform's uses, we describe:

- Accessing services on a remote system over the Web
- Making a Car.com service available for use by external systems
- Accessing a service provided by a proprietary ERP system

In reality, an organization can choose to limit Messaging Platform use to only one of the three scenarios, use two of the three scenarios, or implement all three scenarios. In other words, you can make your enterprise as open as possible or restrict inbound or outbound access.

Accessing Remote Services over the Web

Even though the insurance quotes come from various companies with many different systems, Car.com can interact with them because Car.com has agreed to use their XML schemas. These schemas define, among other things, the content and encoding requirements of a quote request and an actual quote. (For more information on schemas and their use, refer to the sidebar titled "What Are Schemas?")

Car.com customers can request and receive car insurance quotes over the Internet. To provide this service, Car.com uses Messaging Platform to access the quote services of various car insurance vendors.

1. Car.com implements the schema developed by the insurance company for the transmission of quote requests and quotes.
2. A user logs onto the Car.com website and requests a quote.
3. Car.com processes the data and creates a Quote Request object.
4. The Quote Request object is passed to Messaging Platform.
5. Messaging Platform transforms the Quote Request object into a (SOAP-encoded) XML document containing correctly encoded parameters as defied by the agreed-upon schema.
6. Messaging Platform sends the Quote Request XML document to the insurance company.
7. The insurance company receives the document and transforms it into a Quote Request that their system can use.
8. The insurance company processes the Quote Request and returns a Quote, formatted as an XML document. This is sent back to Car.com via Messaging Platform.

What Are Schemas?

A schema is a model for describing the structure of information. The term is borrowed from the database world to describe the structure of data in relational tables. In the context of XML, a schema describes a model for an entire class of documents. The model describes the possible arrangement of tags and text in a valid document. A schema might also be viewed as an agreement on a common vocabulary for a particular application that involves exchanging documents.

Schemas provide the ability to test the validity of XML documents sent and received over the Web. For example, if you're receiving XML transactions over the web, you don't want to process the content if it's not in the proper schema. Thus, schemas are proving very useful for the e-commerce environment, because they provide a mechanism to exchange accurate data between systems that would otherwise not be able to share. Many industries, such as insurance and telecommunications, are developing standardized schemas.

Schemas are similar to Datatype Definitions (DTD). Messaging Platform uses schemas instead of DTDs because schemas allow you to develop richer structural rules. In addition, there is vast and growing support in the computer industry for the use of XML schemas.

9. Messaging Platform receives the XML Quote document and verifies it against the schema. It then sends the appropriate data from the document to Car.com.
10. Car.com displays the Quote for its customer.

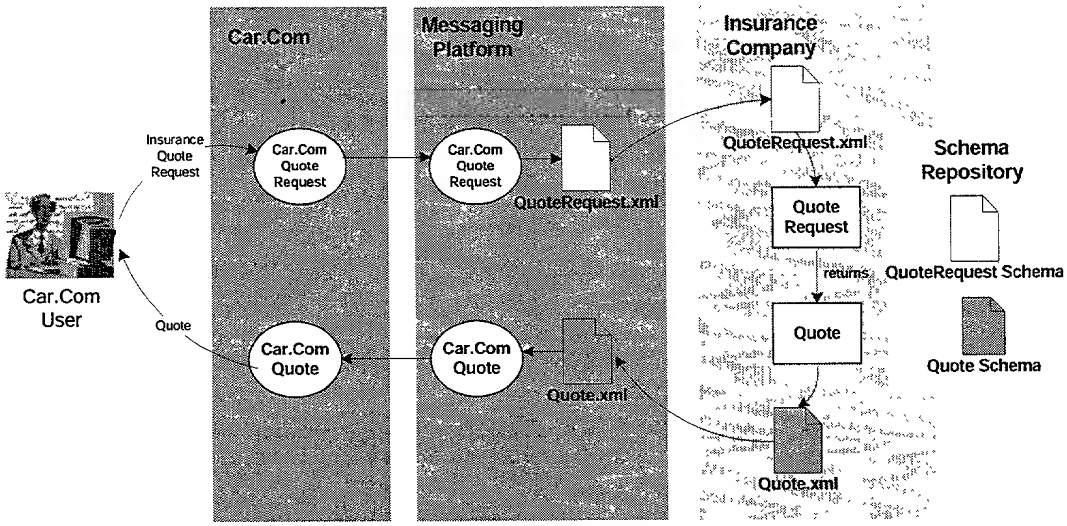


Figure 2. Using Messaging Platform to access services over the Web

Making Services Available over the Web

Car.com stores demographic information about its customers and makes it available for other vendors. Vendors can access information such as home address, age, and driving record and can then use this information to target people in specific age groups or locations for advertising purposes.

1. Car.com develops a schema for the demographic information that it makes available. Insurance vendors interested in purchasing the demographic data from Car.com agree to use this schema and make sure that they communicate with the demographic information services in accordance to the schema rules.

Messaging Platform and Schemas

The use of schemas is key to Messaging Platform. Their existence makes possible the exchange of data between disparate systems. Messaging Platform checks XML documents against the appropriate schema to ensure that the data is formatted correctly. It then culls out the data needed by the recipient to process the request. If the XML document contains parameters not needed by the recipient system, Messaging Platform simply ignores them. Later, Messaging Platform packages the return values for the sender, including all the parameters the sender packaged in the original XML document even though some of those fields were not touched by the recipient. Thus, Messaging Platform ties together two disparate applications, allowing them to use each others' services in spite of their differences.

2. Car.com's demographic service registers with the Messaging Platform Request Processor. Registering with the Request Processor makes the service available to receive data from external sources.
3. Insurance vendors send an XML document containing a request for demographic information to the Messaging Platform Request Processor.
4. The Request Processor forwards the XML document, sending all the data from the document to the Car.com demographic services.
5. The data from the request is processed by the demographic services, which return the requested information to the Request Processor.
6. The Request Processor repackages the returned information as an XML document that complies to the schema and sends it back to the requesting vendor. The vendor is then responsible for processing the XML document in its own way.

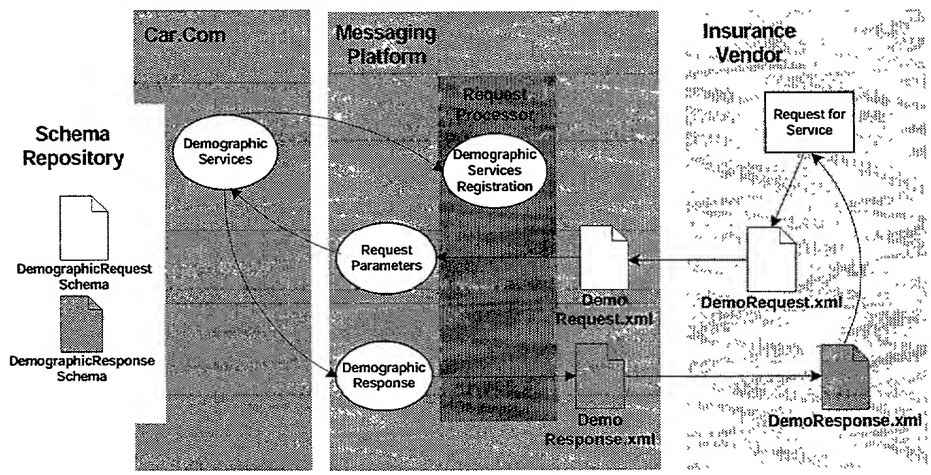


Figure 3. Using Messaging Platform to make services available over the Web

Using Proprietary Services over the Web

Large businesses, such as car manufacturers, often use Enterprise Relationship Planning (ERP) packages to control all aspects of their business, including inventory, billing, and shipping. Since Car.com allows users to order cars directly from manufacturers, Car.com must be able to exchange data with the manufacturer's ERP applications. Many ERPs are proprietary; adapters and connectors are required to link the ERPs with other applications. In this scenario, we describe how Car.com uses Messaging Platform to interact with an adapter and connector to access a car manufacturer's ERP.

1. A Car.com customer orders a car through Car.com. Among other things, he needs to find out when the car will be available for delivery.
2. Car.com processes the order and sends it to Messaging Platform for transmission to the car manufacturer.
3. Messaging Platform creates an XML Order document containing the purchase information. The XML document is formatted according to the rules of a schema published by the car manufacturer.
4. Messaging Platform sends the XML document to the adapter.
5. The adapter converts the information sent from Messaging Platform into a format that the connector can work with. (For example, it might convert the information into another XML document formatted differently than the one sent by Car.com.)
6. The adapter sends the request to a connector that has been developed specifically for the ERP. The connector parses the information in the request and sends it to the ERP, which then processes the order and returns requested information, such as a delivery date for the new car.
7. The connector sends the returned information to the adapter, which repackages the returned information as an XML Delivery Date document. Again, the XML document is formatted according to the manufacturer's schema.
8. The adapter sends the XML Delivery Date document to Messaging Platform. Messaging Platform processes the XML document and sends the Delivery Date data to Car.com.
9. Car.com processes the Delivery Date information and then displays the information for its customer.

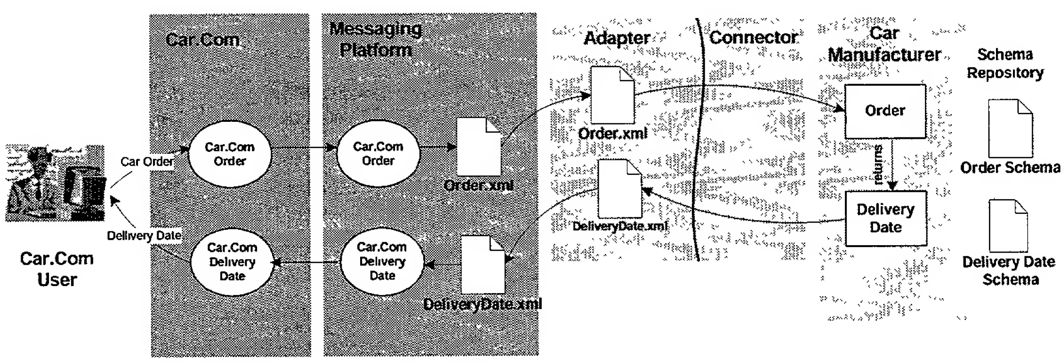


Figure 4. Using Messaging Platform to access proprietary services over the Web

Summary of Scenarios

The scenarios presented in this section describe a number of ways that Messaging Platform can be used within and across enterprises to connect and communicate between heterogeneous systems. In the scenarios, the sender and recipient remain quite decoupled, even when "connected" via Messaging Platform. This decoupling allows the sender and recipient to be constructed independently, glued together only by the agreement of a schema between them.

Messaging Platform provides a business object API to developers of the sender and recipient, shielding the intricate and error-prone construction of XML documents from these developers. This level of abstraction ensures a high level of productivity from a development team.

Tools, Components, and Frameworks

Messaging Platform provides functionality in four areas: connection management, protocol management, message structure, and transformation services. For each area, Messaging Platform provides various tools, components, and frameworks that simplify its use and implementation.

Connection Management. Messaging Platform provides a tool called Connection Assembler that developers use to create, manage, and manipulate Messaging Platform connections. These include connections from external systems to Messaging Platform and from Messaging Platform to external systems.

Protocol Management. Messaging Platform provides a framework for the implementation of Hypertext Transfer Protocol (HTTP) for transport. Although Messaging Platform uses HTTP as its default transfer protocol, open interfaces are provided to help developers implement the protocol of their choice.

Message Structure. Messaging Platform provides a tool called Schema Generator that developers can use to create XML documents containing the information to be transported to another system. Schema Generator also greatly simplifies the creation of the XML schema, which is a document used by systems to "read" the XML documents.

Messaging Platform also provides an Encoding Component that converts XML documents into encoded objects that can be understood and used by the system receiving them. Messaging Platform by default uses the Simple Object Access Protocol (SOAP) to encode XML documents as objects.

Transformation. Messaging Platform contains an Extensible Stylesheet Language (XSL) Translation Component that allows users to provide an alternative to using SOAP for encoding objects. For systems that do not use SOAP, XSL provides a way to transmit information in the format that the system understands.

Messaging Platform Tools and Components

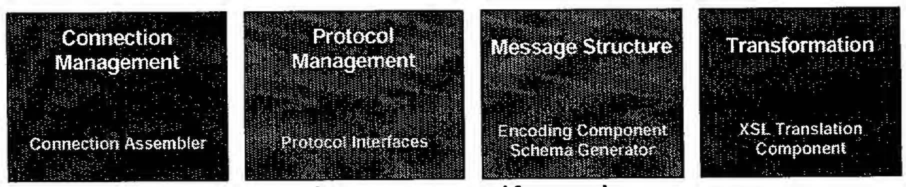


Figure 5. Messaging Platform tools, components, and frameworks

Messaging Platform Features

Messaging Platform's primary benefit is that it provides a way for disparate systems to communicate. This is accomplished through the use of supported and tested standards, such as XML, HTTP, and SOAP. Messaging Platform provides numerous other benefits:

Choice of transport and encoding mechanisms. By default, Messaging Platform uses some specific protocols, such as HTTP for message transport and SOAP for message encoding. However, users can adopt other protocols easily. For example, FTP could be used instead of HTTP and COINS or XMLRPC could be used instead of SOAP.

Flexible XML layout options. Messaging Platform allows users to determine the layout of their XML documents.

Choice of DOM implementation. A Document Object Model (DOM) determines how the information in an XML document is read and processed by the system. Numerous DOM implementations have been created by different companies and organizations. Messaging Platform provides Sun Microsystems' DOM by default; however, you can use any other DOM instead.

Choice of schema. Schemas are documents that describe the information contained in XML documents. Messaging Platform provides you with some default schemas, but does not limit you to their use. Messaging Platform supports any schema of your choice. This means that you can use a standard schema, a standard schema with some customization, or a schema that is unique.

Schema generation. Messaging Platform provides a GUI tool called the Schema Generator that automates the creation of schemas and the serialization of objects. The Schema Generator's GUI provides a simple and quick way to select the fields to be included in the schema and its serialized object, thus making it easy to modify schemas and serialized objects during the development process.

Object neutrality. Messaging Platform accommodates for differences in the way systems "think about" information. For example, a sender system might conceptualize an object that contains customer information as "Customer." However, the recipient system might consider that object to be "User." Despite their differences, these systems can exchange customer information because Messaging Platform transmits data in an object-neutral manner through the use of schemas. As long as the two systems agree to use the same schema, Customer and User objects can exchange data seamlessly.

Support for XSL. Because not everyone will want to encode objects using SOAP, Messaging Platform supports the use of XSL for formatting and presenting complex XML documents. Messaging Platform provides SAXON 4.5 as the XSL engine; however, you can replace it with the XSL processor of your choice.

Inbound Message Process

In Messaging Platform, the processes for receiving inbound messages and sending outbound messages differ. This section explains the process by which Messaging Platform receives inbound messages, or, more specifically, receives a request to use one of its services from an external system. Inbound messages are messages sent from a sender to a recipient via Messaging Platform.

The section starts with a high-level look at the inbound message process. It then provides more detail about how a recipient registers with the Request Processor and how Messaging Platform makes it possible for a sender to invoke a remote system's method.

Inbound Message Process Overview

This section provides a high-level overview of the Messaging Platform's inbound message process. During the inbound process, a sender sends a request to the recipient asking the recipient to run one of its methods and return the result to the sender.

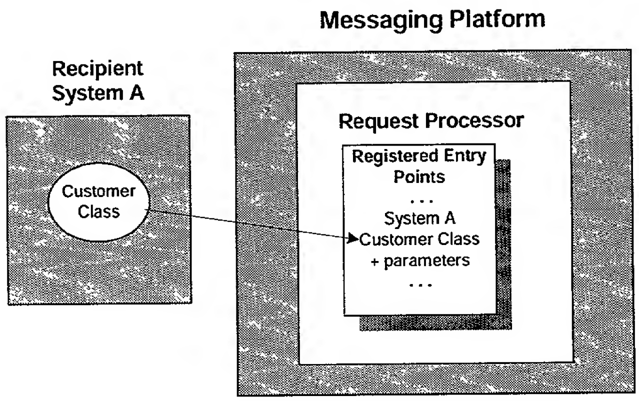
In this section we describe how:

- A recipient registers itself with the Messaging Platform Request Processor
- The sender sends a request to the Request Processor
- The Request Processor extracts information from the request and sends it to the recipient
- The Request Processor sends the recipient's return information back to the original sender

1. The recipient registers its "entry points" with the Request Processor.

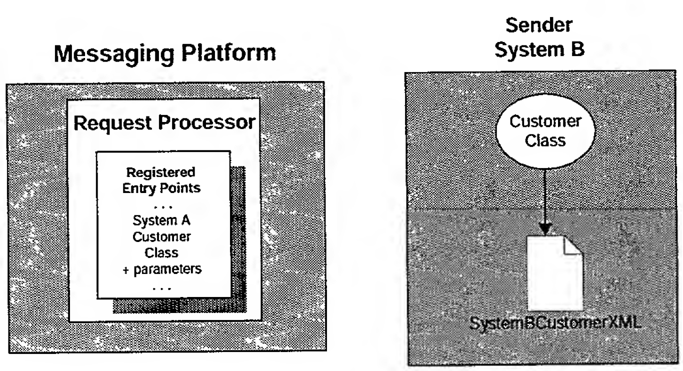
   Entry points are classes that are able to receive information from a remote sender. By registering its entry points, methods in the recipient are made available for calling via XML. Not every class in an application needs to be an entry point.

When a recipient registers, it tells the Request Processor about itself. For example, it registers its methods and the parameters the methods accept. The Request Processor saves this information for future reference.

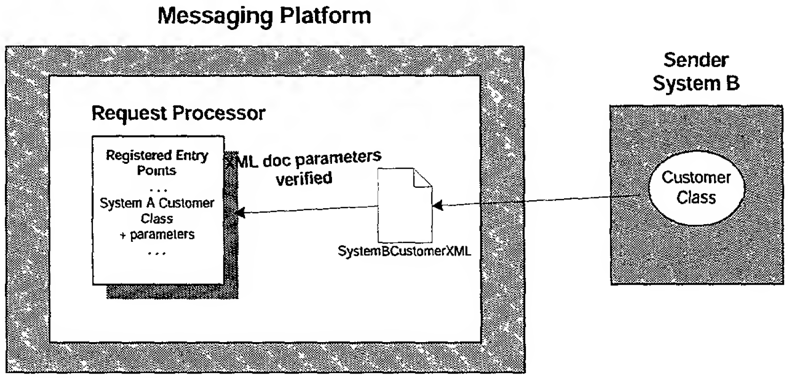


2. The sender creates an XML document containing a request to use one of the recipient's services.

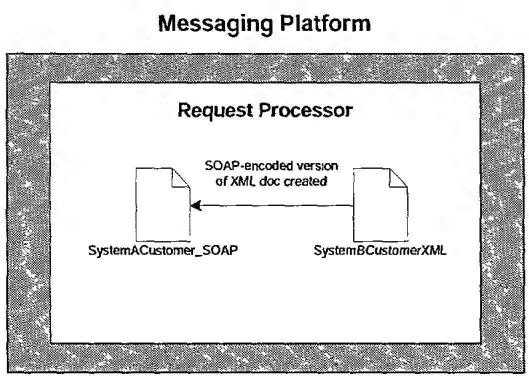
   For example, the sender's Customer class might need to get information about one of the recipient's customers. So, it sends a request to the recipient that will cause the recipient to run one of the methods it registered at the Request Processor to provide the needed information. The parameters to be passed to the method are also contained in the XML document.



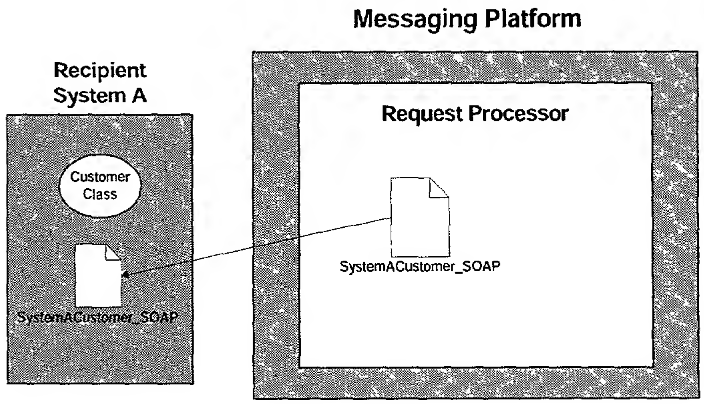
3. The sender sends an XML document to the Request Processor. Upon receiving the XML document, the Request Processor makes sure that the XML document can be accepted by its intended recipient.



4. The Request Processor identifies the method that the sender wishes to execute. The Request Processor then reviews the XML document, determines which parameters in the document are needed by the recipient, and creates SOAP-encoded versions of the parameters.



5. The Request Processor forwards the SOAP-encoded parameters to the recipient. These are recognized by the recipient as objects that implement the appropriate interface for each parameter.



6. The recipient method develops a response, or a return value. This value is sent to Request Processor, which encodes the return information for use by the sender in an XML document
7. Messaging Platform sends the XML document containing the response to the sender.

In the next two sections, "Registration" and "Invoking Methods at the Recipient," the steps presented above are explored in greater detail.

Registration

Although Messaging Platform provides some GUI tools, such as the SchemaGenerator and the ConnectionAssembler, for the most part it is a framework that simplifies the development and implementation of Messaging Platform. Because of this, the descriptions presented in this section and the next are largely descriptions of the functionality of various Messaging Platform classes and their objects.

Note: In this document, we are referring to classes in a generic manner. In reality, the Messaging Platform class names include information about the runtime being supported. For example, in Step 1 below, we refer to a RegistrationRequest object. In reality, this would be called the CPRegistrationRequest object if Component Platform were the runtime in use or EJBRegistrationRequest if the runtime were EJB.

1. The developer uses the Component Assembler to create a new Messaging Platform component. For more information on creating Messaging Platform components, go to Messaging Platform Component Templates on page 29.
2. Using the Launcher Component, the developer launches the component associated with the bean. When the component launches, the registration process is initiated.
3. As part of registration, the recipient's methods and parameters are packaged into a URL. The Request Processor receives the URL and validates it by making sure that all the content is present and in the correct form.
4. The Request Processor creates a RuntimeRequestHandler object for the registered component.

   The RuntimeRequestHandler contains a method called getInvokeable that uses reflection to obtain from the recipient class a complete list of its methods and their parameters. The RuntimeRequestHandler stores this list and the parameters contained in the registration URL for use later when it verifies that methods requested by senders are, in fact, methods that the recipient can run.

Note: There is a one-to-one relationship between registered components (recipients) and RuntimeRequestHandler objects.

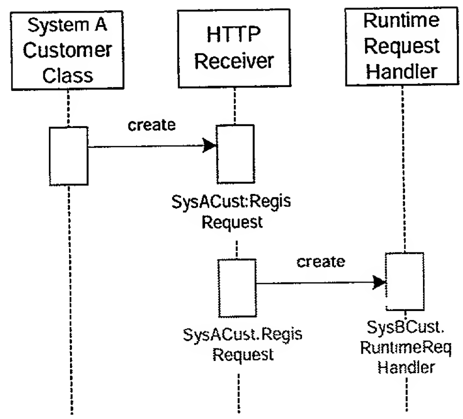


Figure 6. The recipient registration process

Invoking Methods at the Recipient

This section explains in detail the process by which a sender invokes a recipient's method. For clarity, we have separated it into three parts:

- Part One describes the process by which the sender creates an XML document and sends it to the Request Processor.
- Part Two describes the process by which the Request Processor repackages the XML document and sends it to the recipient.
- Part Three describes the process by which the recipient's return values are packaged and sent back to the sender.

Note: This description focuses on Component Platform for EJB.

Part One: From the Sender to the Request Processor

1. The sender uses the MessagingController factory class to create a new MethodCall object.

   The MethodCall object identifies the method to be invoked at the recipient and includes the parameters the sender wishes to pass to the method.

2. The sender uses MessagingController to create an Encoder object.

   The Encoder identifies the type of encoding required by the recipient.

3. The sender uses MessagingController to create a new RuntimeRequest object. The MethodCall and Encoder objects are passed to the new RuntimeRequest.

4. The sender tells RuntimeRequest to run the FireURL method. At this point, the RuntimeRequest object tells the Encoder to create a message containing the method and parameter information contained in the

MethodCall object. This message is encoded in the manner required by the recipient, which is specified by the Encoder object. (Since SOAP is the default Messaging Platform encoding mechanism, we will consider all encoded XML documents to be SOAP-encoded.)

5. The XML document is then attached to a URL, which the RuntimeRequest object sends to the Request Processor.

   The URL created in this step contains information that matches part of the URL sent to the Request Processor by the recipient during its registration process. The information that the recipient sent in the URL at registration is saved in the recipient's unique RuntimeRequestHandler, which is stored by the Request Processor.

   The URL contains the following information: http://server/mode-id/encoding-id/url + XML document.

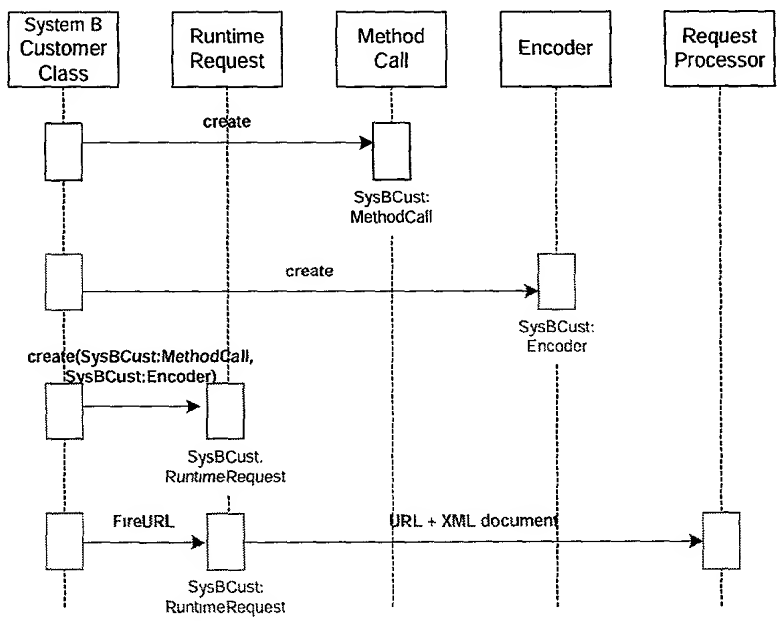


Figure 7. XML document is sent from the sender to the Request Processor

Part Two: From the Request Processor to the Recipient

1. The Request Processor receives the URL from the sender.
2. The Request Processor identifies the name of the RuntimeRequestHandler to be used for this request.

   The name of the RuntimeRequestHandler corresponds to the URL-ID contained in the URL.

3. The Request Processor determines the type of ContentHandler that must be created for the request. The type of ContentHandler needed is determined by the Encoding-ID contained in the URL. The Request Processor then makes a new ContentHandler object for the specified Encoding-ID.
4. When the XML document is received by the Request Processor from the network, the document is no more than a stream of bytes. This stream needs to be converted into an object that Messaging Platform can use. To do this, the Request Processor tells the ContentHandler to run its getContent method. This converts the stream of bytes into the appropriate object. For example, if SOAP is the encoding type, the appropriate object is a SOAP-encoded XML document.
5. The Request Processor tells the RuntimeRequestHandler to get its Invokeable and then sends the data contained in the XML document created in Step 4 to the Invokeable.
6. The Request Processor tells the Invokeable to run its Invoke method, which in fact runs the method and parameters specified in the data contained in the XML document.

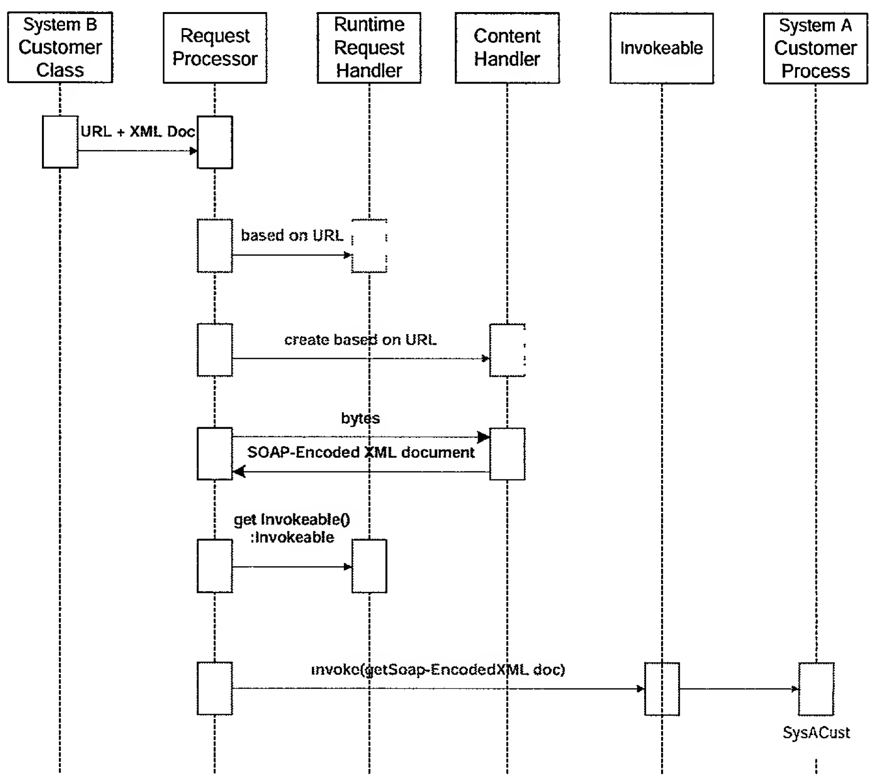


Figure 8. XML document is sent from the Request Processor to the recipient

Part Three: Returning Values to the Sender

When the invoked method runs, it returns a value. This value is returned to the sender via the Request Processor.

1. The invoked method may return a value, which is sent to the Request Processor.
2. The Request Processor sends the return value to the ContentHandler that was created when the original message was sent.
3. The ContentHandler contains a method called formResult, which takes the return value and returns it in the particular manner required by the sender (for example, SOAP-encoded).
4. The ContentHandler sends the returned object to the Request Processor.
5. The Request Processor sends the return value to the sender.

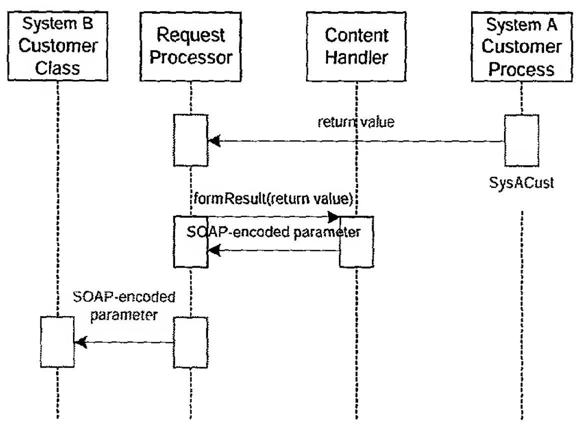


Figure 9. Return value is sent to sender

Outbound Message Process

This section explains Messaging Platform's process for sending messages from Component Platform for EJB to a recipient whose runtime is unknown. The outbound process allows you to send a request for services to an external system.

The outbound message process is considerably simpler than the inbound message process because of the Connection Assembler. The Connection Assembler is a GUI tool that automates the process of identifying and storing the addresses of remote destinations.

Note: This is the process for sending messages without the Connection Assembler. This document will be revised in the near future to include the Connection Assembler procedure instead.

Outbound Message Process

This section describes the process by which a sender sends information to a recipient that resides on a different system. It assumes that you are developing Component Platform components or other Java applications. The recipient can be any other system.

1. A connection information file is created for the recipient. This is an XML document that defines the:
   - Host on which the recipient resides (server)
   - Port on which the recipient resides
   - Mode-ID
   - Encoding-ID

   The name of this file becomes synonymous with the name of the connection.
2. The file is saved to [Component Repository]_Common\Messaging Runtime\Connections\Outbound directory.
3. A Connection object is created and passed the name of the connection that was specified in Step 1. (The name of the XML file is the name of the connection.)

4. A RuntimeRequest object is created and passed the recipient's URL and the method call to be invoked at the recipient.
5. The RuntimeRequest object runs the FireURL method. FireURL sends information encapsulated in the RuntimeRequest object to the Request Processor in a URL.
6. The sender may receive a return value from the recipient. If a return value is received, it is sent to the RuntimeRequest's getResponse method. RuntimeRequest creates an Encoder based on the type of encoding used by the returned document.

Messaging Platform Component Templates

This section briefly introduces Assembler templates and then provides you with information about each Messaging Platform templates. It explains what information you must provide for each template and what items the Assembler provides for you in return.

Assembler Templates

Component Platform provides a tool called the Component Assembler that you use to manage and configure components. The Assembler offers a number of predefined templates that you use to create new components. Templates provide you with a starting point for component creation. They provide you with the component's directory structure, default configurations, and some basic classes to which you add additional code to further develop the component. Since different templates create different types of components and runtime environments, the items provided by the Assembler differ for each template.

Five Messaging Platform components are available from the Assembler: CR Inbound, CR Outbound, J2EE Inbound, J2EE Outbound, and Adapter.

Note: For more information about Component Platform, refer to the *Component Platform Overview* guide.

For general information on using the Assembler, refer to the first two sections of *Developing WebLogic Applications with Component Platform* guide.

Component Runtime Inbound Template

The Component Runtime Inbound template creates a Component Runtime (CR) component that uses Messaging Platform to accept incoming requests. This component contains a service that can receive requests from external systems. It also contains other elements, such as a Request Processor and Launch Items, that are needed for inbound

messaging to be successful. Table 1 lists the parameters that you provide to the CR Inbound template to create a Messaging Platform Inbound Request component.

Table 1. Parameters Required by the CR Inbound Template

Parameter	Description
Request Processor Name	Name of the Request Processor.
Request Processor Port	The port on which the Request Processor listens.
Recipient Name	Name of the recipient. "Recipient" refers to the service that accepts requests from external systems.
Recipient Alias	The name by which Component Platform recognizes the recipient. This also is the recipient's URL address.
Recipient Type	A way to classify the recipient (for example, "CR Recipient"). The Explorer component uses this information to sort recipients.

The Assembler creates a new CR Inbound component from the information you provided. The component is a "skeleton" component that provides with you a solid starting point for further component development. Table 2 lists the items created for the new component.

Table 2 Items Created by the Assembler for CR Inbound Components

Item	Description
Request Processor process	A Component Platform server process that contains two named objects that you can run: the Request Processor and the Recipient.
Sender process	A Component Platform client process that submits requests to the Recipient through the Request Processor. This is provided for testing purposes.
Global, client, and server requirements	Other components or files required by the CR Inbound component.
Processor Launch Item	Launches the Request Processor and the Recipient.
Sender Launch Item	Launches the Sender GUI.
Source code	[Component Repository]\[Component Version]\src\[component_name]: \classes: [Recipient Name]Request.java = Packages a request as XML and then runs the FireURL method to send a request to the external service. In Inbound Messaging Platform processes, the Sender uses this class to package and send requests. \facilities: IDEStartup.java = Provides code to run compiled component \html InboundTest.html = Provides a Web page from which you can submit a request. Available for testing purposes.

Item	Description
	\inbound: Constants.java = Constants used by the component **Note:** The next three classes, working together, form a named object, specifically, a Recipient. This type of class grouping—an interface, a default implementation, and a stub—is used throughout Component Platform to create named objects. [Recipient]Recipient.java = Interface for the Recipient. [Recipient]RecipientImpl.java = Default implementation for the Recipient [Recipient]RecipientImpl_Stub.java = Stub for the Recipient **Note:** The next three classes, working together, form a Request Processor. [Request Processor]Processor.java = Interface for the Request Processor. [Request Processor]ProcessorImpl = Default implementation for the Request Processor. [Request Processor]ProcessorImpl_Stub.java = Stub for the Request Processor. \windows: [Recipient]InboundClientFrame.java = Creates a GUI for the Sender. This is available for testing purposes.

Further Developing the CR Inbound Component

The code provided by the Assembler, when compiled, provides you with enough material to run the component's processes. However, further development is, of course, required to add your desired functionality to the component.

Specifically, you will add code to the [Recipient]Recipient group of classes to create functionality. You do not need to make changes to any of the Request Processor classes; the Assembler creates a Request Processor that is fully functional.

Component Runtime Outbound Template

The Component Runtime Outbound template creates a Component Runtime (CR) component that uses Messaging Platform to send outgoing requests. This component contains a Sender process that the Sender uses to send a request to an external service. Table 3 lists the parameters that you provide to component.

Table 3. Parameters Required by the CR Outbound Template

Parameter	Description
Connection Name	Name of the Connection. Connection information is saved in an XML document that will also have this name.
Destination Host	The name of the host on which the Recipient resides.
Destination Port	The Recipient's port.
Destination URL	The Recipient's URL. This is also the name by which the Request Processor recognizes the Recipient; it must match the Recipient's alias provided when it was created using the Inbound template.
Destination Method	The Recipient method to be invoked by the request.
Destination Protocol	List of protocols – select the protocol used by the Recipient.
Destination Encoding	List of encoding options – select the encoding option used by the Recipient.

The Assembler creates a new CR Outbound component from the information you provided. The component is a "skeleton" component that provides with you a solid starting point for further component development. Table 4 lists the items created for the new component.

Table 4 Items Created by the Assembler for CR Outbound Components

Item	Description
Sender process	The process that sends request to an external service.
Global, client, and server requirements.	Other components or files required by the CR Outbound component.
Sender Launch Item	Launches the Sender GUI.
Source code	[Component Repository]\[Component Version]\src\[component_name]: \classes: [Component]Request.java = Packages a request as XML and then runs the FireURL method to send a request to the external service \facilities: IDEStartup.java =provides code to run compiled component \windows: [Sender]OneOutboundClientFrame.java = Creates a GUI for the Sender. This is available for testing purposes.

Further Developing the CR Outbound Component

The code provided by the Assembler, when compiled, provides you with enough material to run the component's processes. However, further development is, of course, required to add your desired functionality to the component.

You will probably extend the [Component]Request.java class so that it requests different or additional services. You may also extend the Sender, although it is more likely that you will create an entirely new Sender, using the one provided by the Assembler only for testing purposes.

Adapter Template

The Adapter template creates a Component Runtime component that uses Messaging Platform to send requests to and receive requests from a external system that is not using Messaging Platform. This component contains an Adapter process that listens for requests coming from external, proprietary systems. The Adapter does not register with a Request Processor; instead, it actually functions much as a request processor does, as it accepts requests and routes them to the correct service.

The component also contains a Sender process that the Sender uses to send a request to a proprietary system. Table 5 lists the parameters that you provide to the Adapter template to create a Messaging Platform Adapter component.

Table 5. Parameters Required by the Adapter Template

Parameter	Description
Adapter name	Name of the adapter. Connection information for the adapter in an XML document that will also have this name.
Port	Port on which adapter is listening for requests.

The Assembler creates a new Adapter component from the information you provided. The component is a "skeleton" component that provides with you a solid starting point for further component development. Table 6 lists the items created for the new component.

Table 6 Items Created by the Assembler for Adapter Components

Item	Description
Adapter process	The process that starts running the Adapter.
Sender process	The process that sends request to an external service via the Adapter.
Global, client, and server requirements.	Other components or files required by the Adapter component.
Adapter process Launch Item	Launches the Adapter.
Sender Launch Item	Launches the Sender GUI.
Source code	[Component Repository]\[Component Version]\src\[component_name]: \adapter: Constants.java = Constants used by the component [Adapter]AdapterConnectionImpl.java = Provides code for the Connector to which the Adapter sends requests **Note:** The next three classes, working together, form a named object, specifically, an Adapter. This type of class grouping—an interface, a default implementation, and a stub—is used throughout Component Platform to create named objects. [Adapter]Adapter.java = Interface for the Adapter [Adapter]AdapterImpl.java = Default implementation for the Adapter [Adapter]AdapterImpl_Stub.java = Stub for the Adapter \classes: [Component]Request.java = Packages a request as XML and then runs the FireURL method to send a request to the external service \facilities: IDEStartup.java =Provides code to run compiled component \html: AdapterTest.html = Provides a Web page from which you can submit a request that is sent via the Adapter. Available for testing purposes. \windows: [Adapter]OneClientFrame.java = Creates a GUI for the Sender. This is available for testing purposes.

Further Developing the Adapter Component

The code provided by the Assembler, when compiled, provides you with enough material to run the component's processes. However, further development is, of course, required to add your desired functionality to the component.

Specifically, you will add code to the [Adapter]AdapterConnectionImpl.java class. This class is, in fact, the Connection portion of the Adapter/Connector, and will need to be modified so that it can interact with an external system.

J2EE Inbound Template

The J2EE Inbound template creates an EJB component that uses Messaging Platform to accept incoming requests. This component contains a Bean that can receive requests from external systems. It also contains a process that allows you to monitor requests. Table 7 lists the parameters that you provide to the J2EE Inbound template to create a Messaging Platform Inbound Request component.

Table 7. Parameters Required by the J2EE Inbound Template

Parameter	Description
Home	Directory location of WebLogic.
Server Name	WebLogic server.
Server Host	Host on which the WebLogic server resides. The default value is localhost.
Server Port	Port on which the WebLogic server listens. The default value is 8001.
Initial Context Factory Class	The WebLogic initial context factory class. The default is weblogic.jndi.WLInitialContextFactory.
Provider URL	The server URL. The default is t3://localhost:8001
Security Principle	The server username. This value is optional.
Security Credential	The server password. This value is optional.
Bean Name	A name for the Bean to be created.
Bean Type	Currently, Messaging Platform supports only Stateless Beans.

The Assembler creates a new J2EE Inbound component from the information you provided. The component is a "skeleton" component that provides with you a solid starting point for further component development. Table 8 lists the items created for the new component.

Table 8 Items Created by the Assembler for J2EE Inbound Components

Item	Description
GUI Process	A Component Platform process that submits requests to the recipient. This is provided for testing purposes.
Monitor process	A Component Platform server process that provides a Receiver Monitor to monitor requests.
Server process	A WebLogic Runtime process deployment server process. This server launches its Beans and the Request Processor and then registers the Beans with the Request Processor
Global, client, and server requirements	Other components or files required by the J2EE component.
Inbound Test Console Launch Item	Launches the WebLogic Console.
Inbound Test GUI Launch Item	Launches the GUI Process.
Inbound Test Monitor Launch Item	Launches the Receiver Monitor as part of the Explorer Component.
Inbound Test Startup Launch Item	Launches the WebLogic deployment server.
Source code	[Component Repository]\[Component Version]\src\[component_name]: \beans: [bean].java = File created for the stateless Bean. This is a J2EE standard class. [bean]Bean.java = File created for the stateless Bean. This is a J2EE standard class. [bean]Home.java = File created for the stateless Bean. This is a J2EE standard class. \classes: Request.java = Packages a request as XML and then runs the FireURL method to send a request to the external service. In Inbound Messaging Platform processes, the Sender uses this class to package and send requests. \facilities: IDEStartup.java = Provides code to run compiled component \services: [server name]MonitorImpl.java = Creates a Receiver Monitor for the component. \windows: InboundClientFrame.java = Creates a GUI for the Sender. This is available for testing purposes.

Further Developing the J2EE Inbound Component

The code provided by the Assembler, when compiled, provides you with enough material to run the component's processes. However, further development is, of course, required to add your desired functionality to the component.

Specifically, you will add code to the Beans to create functionality. You will also update Request.java to map it to your Beans. You can also make changes to InboundClientFrame.java to improve its usefulness for your testing purposes. You do not need to make changes to MonitorImpl.java; the Assembler creates a Receiver Monitor that is fully functional.

J2EE Outbound Templates

The J2EE Outbound template creates a Component Runtime component that uses Messaging Platform to send outgoing requests. This component contains Beans that send requests to external services. Table 9 lists the parameters that you provide to component.

Table 9. Parameters Required by the J2EE Outbound Template

Parameter	Description
Home	Directory location of WebLogic.
Server Name	WebLogic server.
Server Host	Host on which the WebLogic server resides. The default value is localhost.
Server Port	Port on which the WebLogic server listens. The default value is 8001.
Initial Context Factory Class	The WebLogic initial context factory class. The default is weblogic.jndi.WLInitialContextFactory.
Provider URL	The server URL. The default is t3://localhost:8001
Security Principle	The server username. This value is optional.
Security Credential	The server password. This value is optional.
Destination Host	Host on which the recipient resides.
Destination Port	Port on which the recipient server listens.
Destination URL	The URL of the recipient server.
Destination Method	The method to be invoked at the recipient.
Destination Protocol	The protocol used by the recipient. Select the protocol from the list.
Destination Encoding	The encoding used by the recipient. Currently, SOAP is the only encoding option.
Bean Name	Name of the Bean.
Bean Type	Currently, only Stateless Beans can be created for MP.

The Assembler creates a new J2EE Outbound component from the information you provided. The component is a "skeleton" component that provides with you a solid starting point for further component development. Table 10 lists the items created for the new component.

Table 10. Items Created by the Assembler for J2EE Outbound Components

Item	Description
Server process	A WebLogic Runtime process. This is a deployment server that launches the Beans.
Monitor process	A Component Platform server process that provides a Receiver Monitor to monitor requests.
Global, client, and server requirements.	Other components or files required by the J2EE component.
Outbound Test Console Launch Item	Launches the WebLogic Console.
Outbound Test Monitor	Launches the Receiver Monitor as part of the Explorer Component.
Outbound Test Startup	Launches the WebLogic deployment server.
Source code	[Component Repository]\[Component Version]\src\[component_name]: \beans: [bean].java = File created for the stateless Bean. This is a J2EE standard class. [bean]Bean.java = File created for the stateless Bean. This is a J2EE standard class. [bean]Home.java = File created for the stateless Bean. This is a J2EE standard class. \classes: Request.java = Packages a request as XML and then runs the FireURL method to send a request to the external service \services: [server name]MonitorImpl.java = Creates a Receiver Monitor for the component.

Further Developing the J2EE Outbound Component

The code provided by the Assembler, when compiled, provides you with enough material to run the component's processes. However, further development is, of course, required to add your desired functionality to the component.

Specifically, you will add code to the Beans to create functionality. You might also need to update Request.java. You do not need to make changes to MonitorImpl.java; the Assembler creates a Receiver Monitor that is fully functional.

Monitoring Tools

This section describes the Messaging Platform's monitoring tools. Currently, Messaging Platform provides a tool that allows you to monitor the Request Processor. In the future, additional monitoring tools will be available.

Monitoring the Request Processor

Two tools are available to help you monitor incoming requests received by the Request Processor: the GUI Receiver Monitor and the XML Monitor. Both tools report the number of requests received at the Request Processor and the number of responses it returned to the sender; the monitors also indicates the size of each message, in bytes.

The two monitoring tools provide you with the same information. The GUI Receiver Monitor is a Component Platform component. The Receiver Monitor supports Component Runtime and Enterprise Java Bean runtimes. It can be customized to support other runtimes as well. The XML tool can be viewed from Microsoft's Internet Explorer, version 5.0. The XML tool provides a way to monitor the Request Processor from a remote location.

Using the Request Processor Monitoring Tools

This section describes, at a high level, how to access and use the Request Processor monitoring tools.

Note: Both descriptions assume that you are using a WebLogic server with Messaging Platform. The actual procedure with your application server may differ slightly.

The Receiver Monitor

To use the Receiver Monitor (when using WebLogic):

1. Start the ObjectSpace.
2. Start the WebLogic server.
3. Start the Component Explorer.
4. Start the Receiver Monitor.

5. Run the application that will be receiving messages. Make sure that the application registers its services with the Request Processor, and then initiate a transaction. The viewer displays the request history.

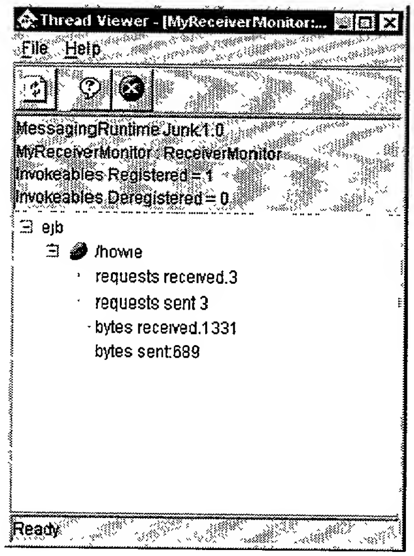


Figure 10. The Receiver Monitor

In Figure 10, only one service (invokeable) is registered at the request processor. If multiple services were registered, the Receiver Monitor would display each service and their track requests.

The XML Monitor

The XML Monitor provides the same information as the Receiver Monitor. When you use the XML Monitor, you must know the invokeable's URL and enter it in the browser's address line. The XML Monitor then takes over, reporting on Request Processor's activity.

Note: The XML Monitor requires Internet Explorer 5.0; it will not work with other Web browsers.

The URL has the following format:

<u>http://[host]:[port]/[mode-ID]/admin</u>

To use the XML Monitor:

1. Start the ObjectSpace.
2. Start the WebLogic server.
3. Run the application that will be receiving messages. Make sure that the application registers its services with the Request Processor, and then initiate a transaction. The viewer displays the request history.
4. Using IE 5.0, enter the URL for the service registered at the Request Processor using MP for the Mode-ID (for example: <u>http://localhost:8001/mp/admin</u>).

Information similar to the following appears in the browser window:

```
    <?xml version="1.0" ?>
-   <mpadmin>
      <registered>3</registered>
      <deregistered>0</deregistered>
    - <modelist>
        <mode>ejb</mode>
      </modelist>
    </mpadmin?
```

When you use MP as the Mode-ID, you receive general information about Request Processor activity. For example, you see that requests have been registered at the request processor and that they are using EJB services. However, this view does not tell you whether or not the requests have been answered, or how large the requests are.

5. For additional information about Request Processor activity, enter the same URL you did in step 4, but substitute the actual Mode-ID for MP (for example, EJB or CP).

   A report similar to the following appears in the browser:

```
    <?xml version="1.0" ?>
-   <modeadmin name="ejb">
    - <invokeable url="/howie">
        <bytessent>1</bytessent?
        <bytesrecieved>1</bytesreceived>
        <requestssent>1</requestssent>
        <requestsreceived>1</requestsreceived>
      </invokeable>
</modeadmin>
```

Appendix A—Schema and DTD for Connection Documents

Appendix A presents the schema and document type definition (DTD) for connection files that you create as part of the configuration process.

Schema for Connection Documents

```
<?xml version="1.0"?>
<!-- XML Schema for Messaging Platform Connection files -->
<schema xmlns="http://www.xpedior.com/mp/ConnectionSchema"
        targetNS="http://www.xpedior.comp/mp/ConnectionSchema"
        version="0.7.5">

<element name="host" type="string" />
<element name="port" type="positive-integer" />
<element name="protocolid" type="string" />
<element name="encodingid" type="string" />
```

DTD for Connection Documents

```
<!-- DTD for Messaging Platform Connection files -->
<!DOCTYPE connection [
  <!ELEMENT host (#PCDATA)>
  <!ELEMENT port (#PCDATA)>
  <!ELEMENT protocolid (#PCDATA)>
  <!ELEMENT encodingid (#PCDATA)>
]>
```

Glossary

ActiveX

A loosely defined set of technologies developed by Microsoft. ActiveX is an outgrowth of two other Microsoft technologies called OLE (Object Linking and Embedding) and COM (Component Object Model).

applet

A program designed to be executed from within another application in the same operating system process. Unlike a applications, applets cannot be executed directly from the operating system. Note that applets are not limited to Java applets running in Web browsers.

bean

See Java Bean

component

A small binary object or program that performs a specific function and is designed in such a way to easily operate with other components and applications. Each Component Platform component is uniquely identified by a version number.

Document Object Model (DOM)

A World Wide Web Consortium (W3C) specification that describes the structure of dynamic HTML and XML documents as a logical structure instead of a collection of tagged words. A DOM defines a document as a tree-like hierarchy of nodes in which the document is an object containing other objects, such as images or forms. Through the DOM API, programs and scripts can access these objects.

Extensible Markup Language (XML)

A condensed form of Standard Generalized Markup Language (SGML). It allows for the creation of customized tags, enabling the definition, transmission, validation, and interpretation of data between applications and between organizations. This differs from HTML tags, which simply specify the format of information.

Extensible Stylesheet Language (XSL)

A language for specifying stylesheets that apply formatting to complex XML documents for presentation in HTML or other formats. Unlike cascading style sheets (CSS), which maps an XML element to a single display object, XSL can map a single XML element to more than one type of display object.

Hypertext Transfer Protocol (HTTP)

A protocol used to request and transmit files, especially Web pages and Web page components, over the Internet or other computer network.

interface

Need a good definition for this

JavaBean

A specification developed by Sun Microsystems that defines how Java objects interact. An object that conforms to this specification is called a JavaBean, and is similar to an ActiveX component. It can be used by any application that understands the JavaBeans format.

The principal difference between ActiveX controls and JavaBeans is that ActiveX controls can be developed in any programming language but executed only on a Windows platform, whereas JavaBeans can be developed only in Java, but can run on any platform.

protocol

An standard agreement for transmitting data between two devices. The protocol determines the type of error checking to be used, the data compression method, if any, how the sending device will indicate that it has finished sending a message, and how the receiving device will indicate that it has received a message.

reflection

A feature of the Java programming language that allows an executing Java program to examine, or "introspect," upon itself, and manipulate internal properties of the program. For example, it's possible for a Java class to obtain the names of all its methods and parameters.

runtime

The base layer of software that enables applications to execute.

schema

A model for describing the structure of information.

serialize

In object-oriented programming, to convert an object into a stream of bytes that can be written out to a file or a disk and later reconstituted into an object.

servlet

An applet that runs on a server. Servlets run within the JVM of the Sun Microsystems Servlet Runner or another server-enabled Web server.

signature

The parameters, such as name and parameters, that are unique to a method and by which it can be identified.

Simple Object Access Protocol (SOAP)

A Remote Procedure Call (RPC) mechanism that is used to transport information across a network. SOAP uses HTTP as the transport mechanism and uses XML documents to encode objects.

XML

See Extensible Markup Language

XSL

See Extensible Stylesheet Language

Index

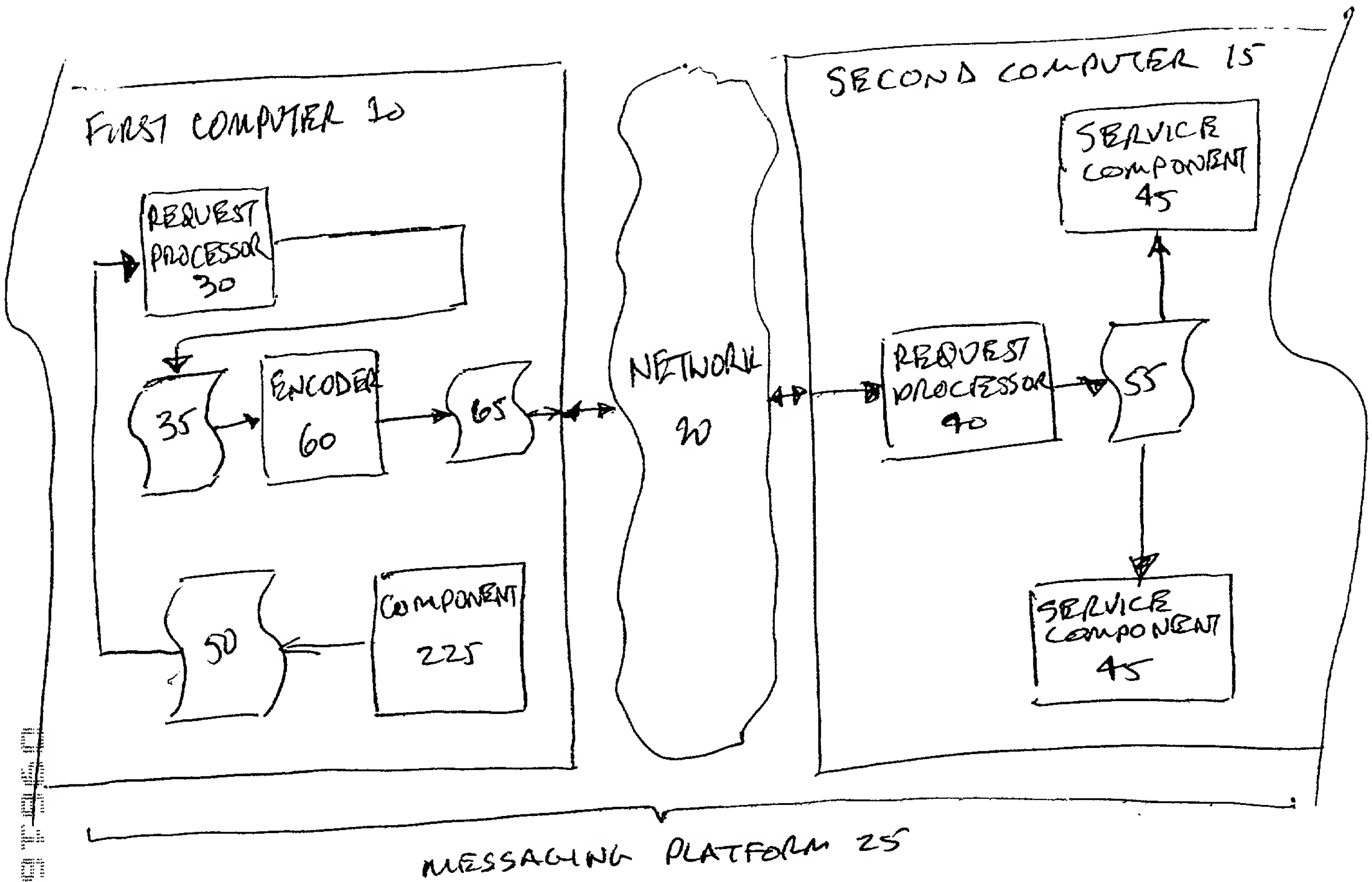

FIRST COMPUTER 10
REQUEST PROCESSOR 30
35
ENCODER 60
65
50
COMPONENT 225
NETWORK 20
SECOND COMPUTER 15
SERVICE COMPONENT 45
REQUEST PROCESSOR 40
55
SERVICE COMPONENT 45
MESSAGING PLATFORM 25

FIG. 1

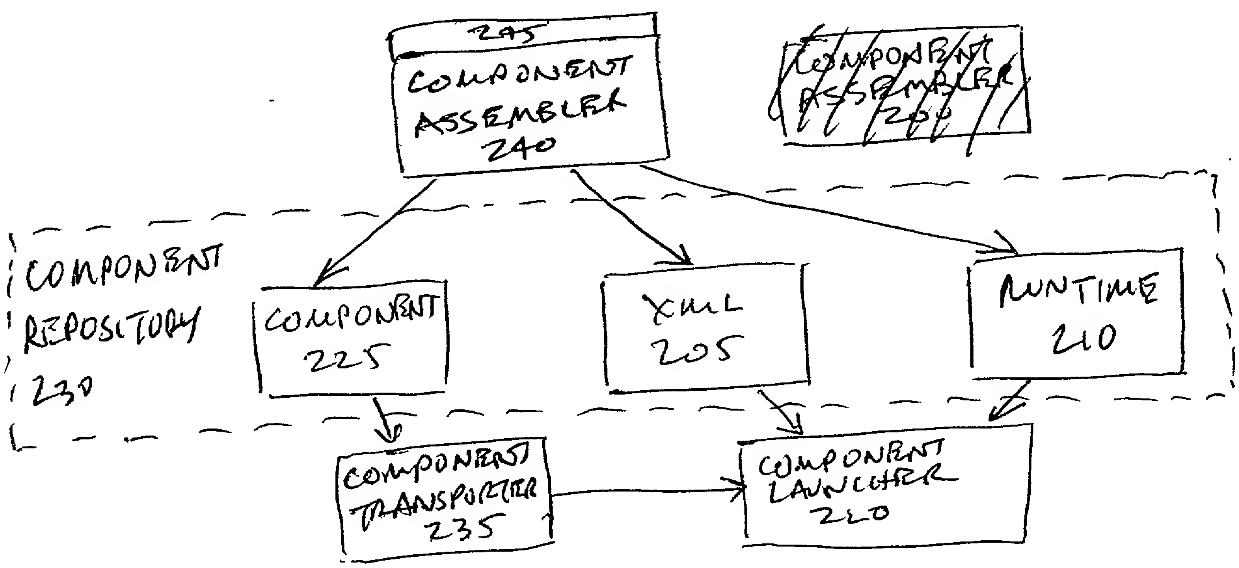

245
COMPONENT ASSEMBLER 240
COMPONENT ASSEMBLER 200
COMPONENT REPOSITORY 230
COMPONENT 225
XML 205
RUNTIME 210
COMPONENT TRANSPORTER 235
COMPONENT LAUNCHER 220

FIG. 3

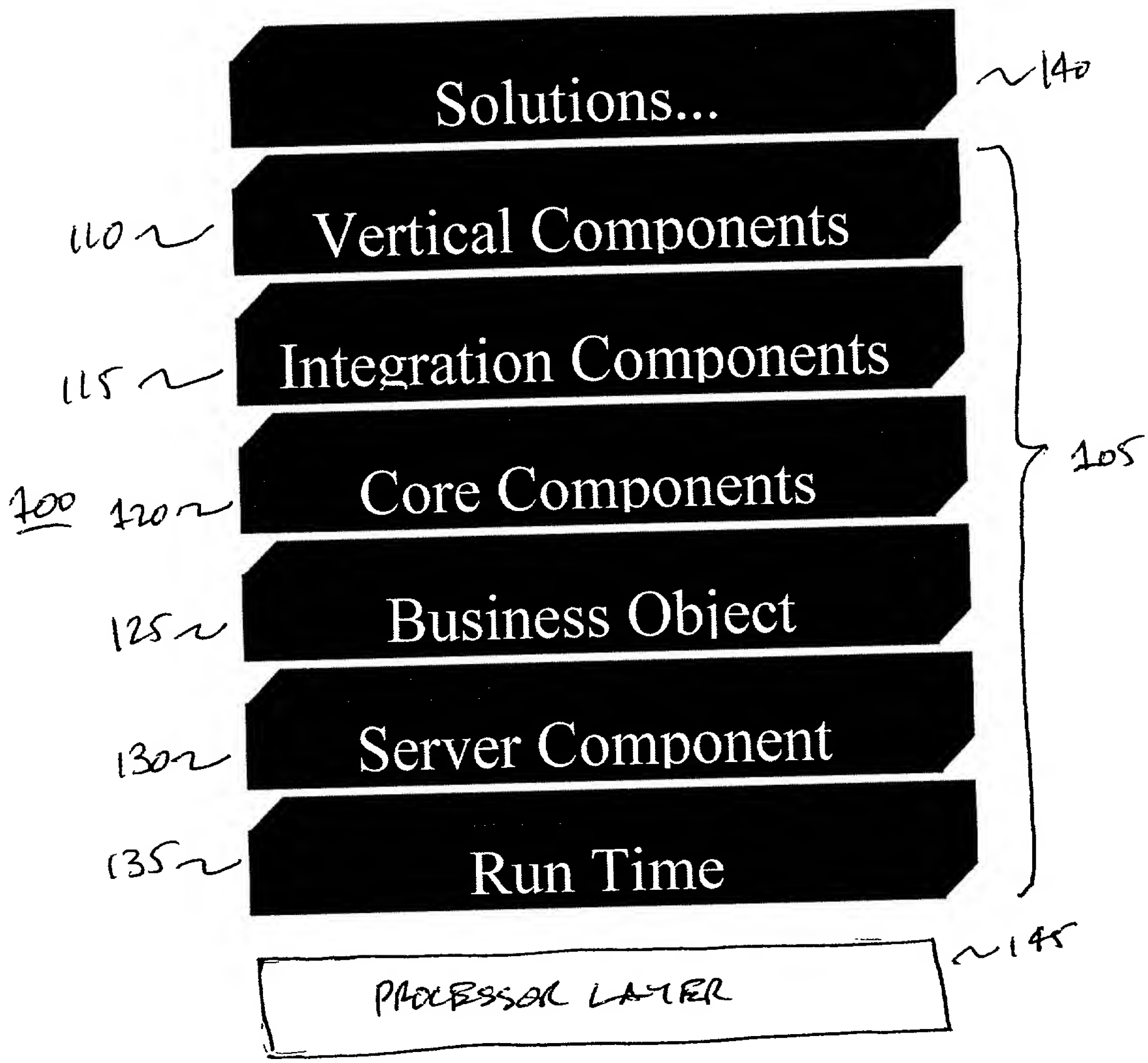
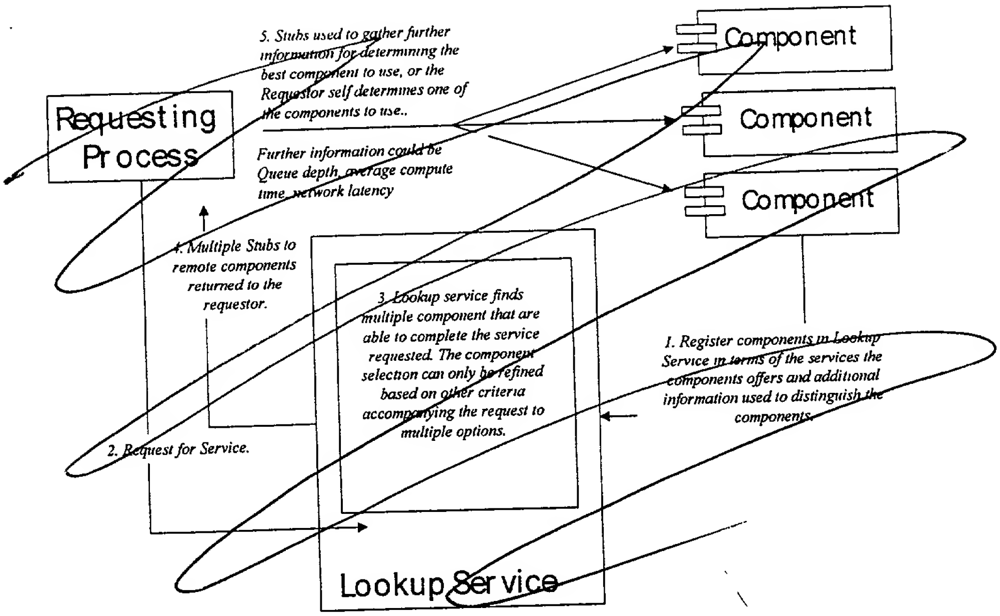
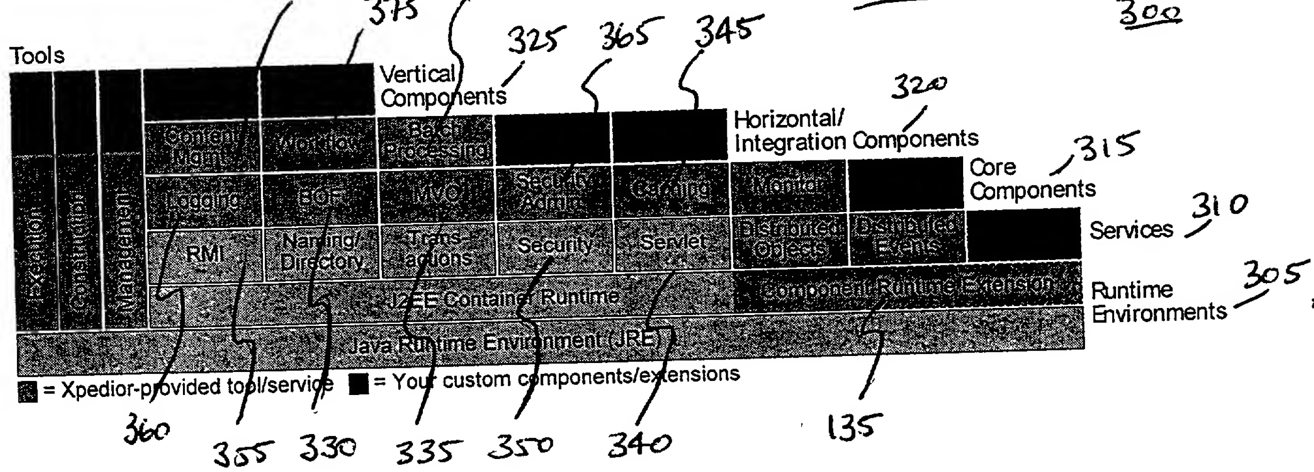

Solutions...
140
Vertical Components
110
Integration Components
115
Core Components
120
100
Business Object
125
Server Component
130
Run Time
135
105
PROCESSOR LAYER
145

FIG. 2



Requesting Process
5. Stubs used to gather further information for determining the best component to use, or the Requestor self determines one of the components to use..
Further information could be Queue depth, average compute time, network latency
Component
Component
Component
4. Multiple Stubs to remote components returned to the requestor.
3. Lookup service finds multiple component that are able to complete the service requested. The component selection can only be refined based on other criteria accompanying the request to multiple options.
1. Register components in Lookup Service in terms of the services the components offers and additional information used to distinguish the components.
2. Request for Service.
Lookup Service

FIG. 4



300
Tools
Vertical Components
Horizontal/ Integration Components
Core Components
Services
Runtime Environments
Execution
Construction
Management
Content Mgmt
Workflow
Batch Processing
Logging
BOF
MVC
Security Admin
Caching
Monitor
RMI
Naming/ Directory
Trans-actions
Security
Servlet
Distributed Objects
Distributed Events
J2EE Container Runtime
Component Runtime Extension
Java Runtime Environment (JRE)
= Xpedior-provided tool/service
= Your custom components/extensions
370
375
380
325
365
345
320
315
310
305
360
355
330
335
350
340
135

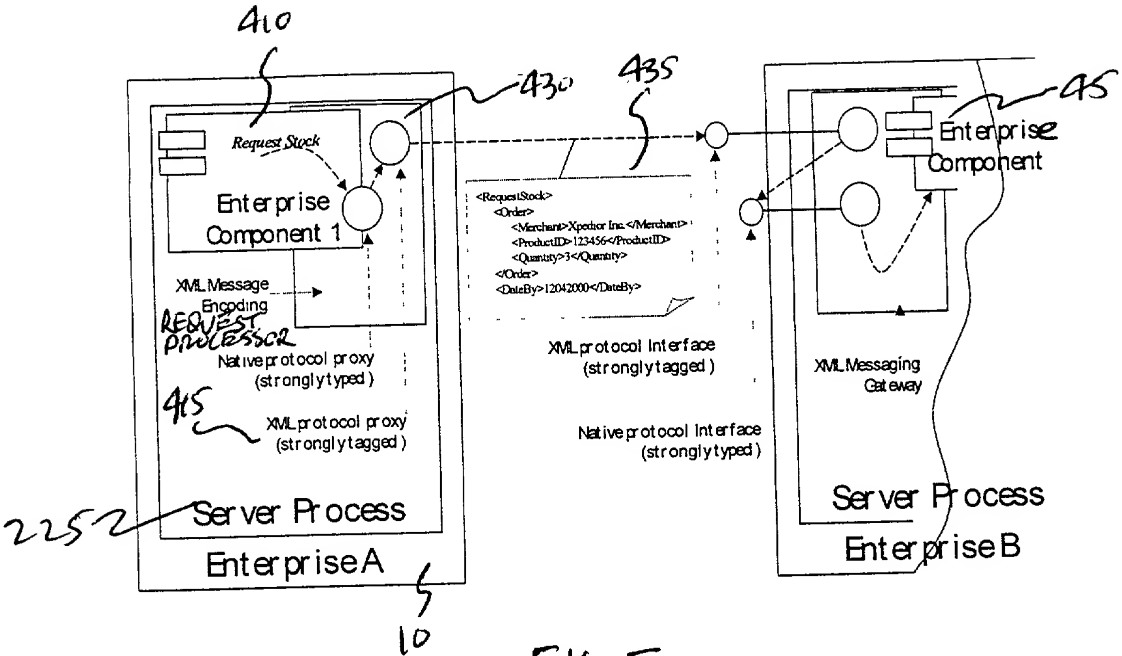

410
430
435
45
Request Stock
Enterprise Component 1
XML Message Encoding
REQUEST PROCESSOR
Native protocol proxy (strongly typed)
415
XML protocol proxy (strongly tagged)
2252
Server Process
Enterprise A
10
<RequestStock>
<Order>
<Merchant>Xpedior Inc.</Merchant>
<ProductID>123456</ProductID>
<Quantity>3</Quantity>
</Order>
<DateBy>12042000</DateBy>
XML protocol Interface (strongly tagged)
Native protocol Interface (strongly typed)
Enterprise Component
XML Messaging Gateway
Server Process
Enterprise B

FIG. 5

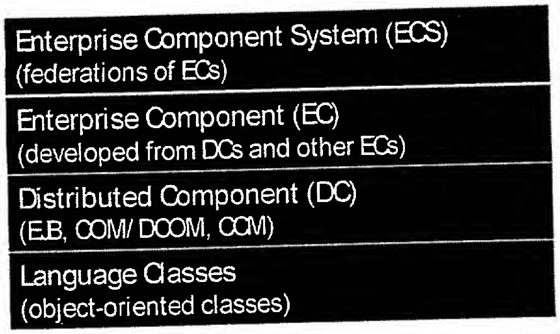


FIG. 8

~~FIG. 2 shows granularity of structures in the preferred embodiment;~~

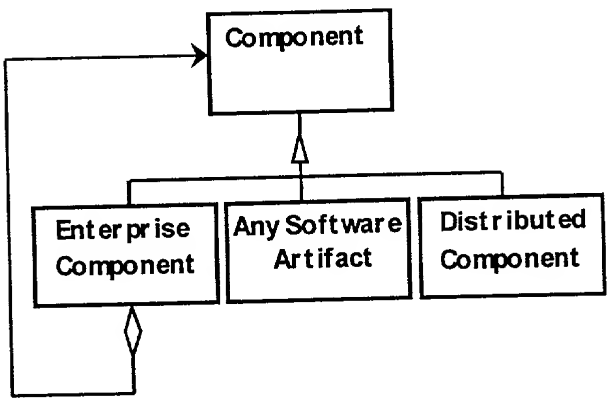


FIG. 6

~~FIG. 3 shows granularity of structures in the preferred embodiment.~~

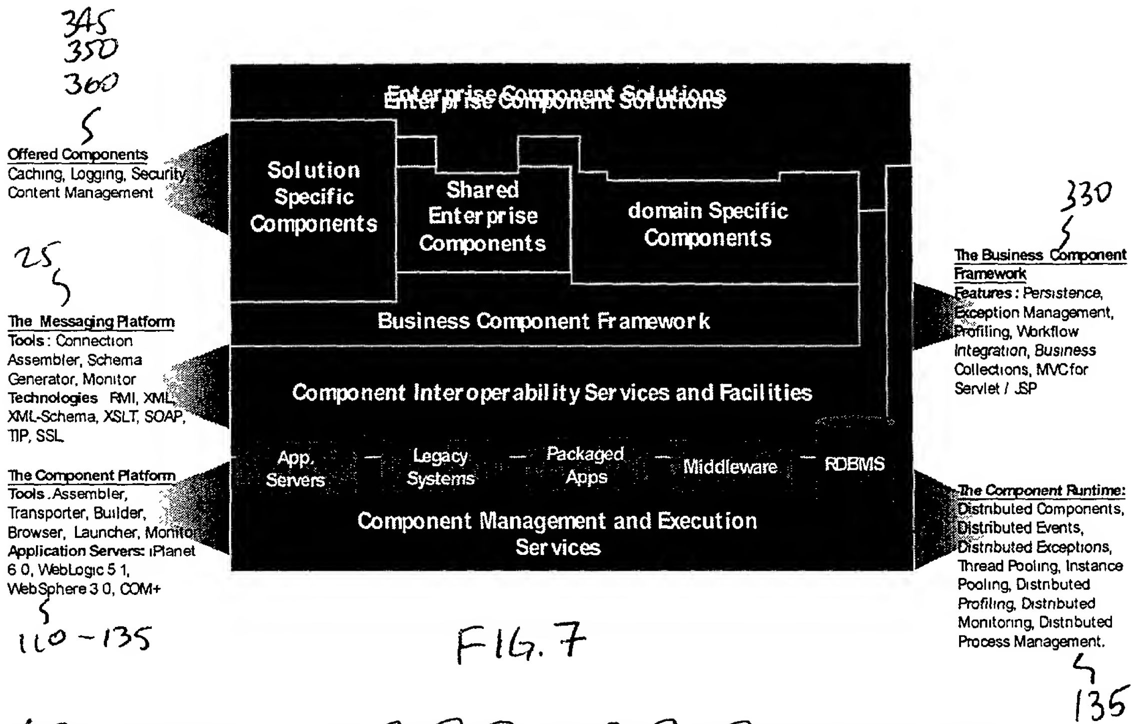
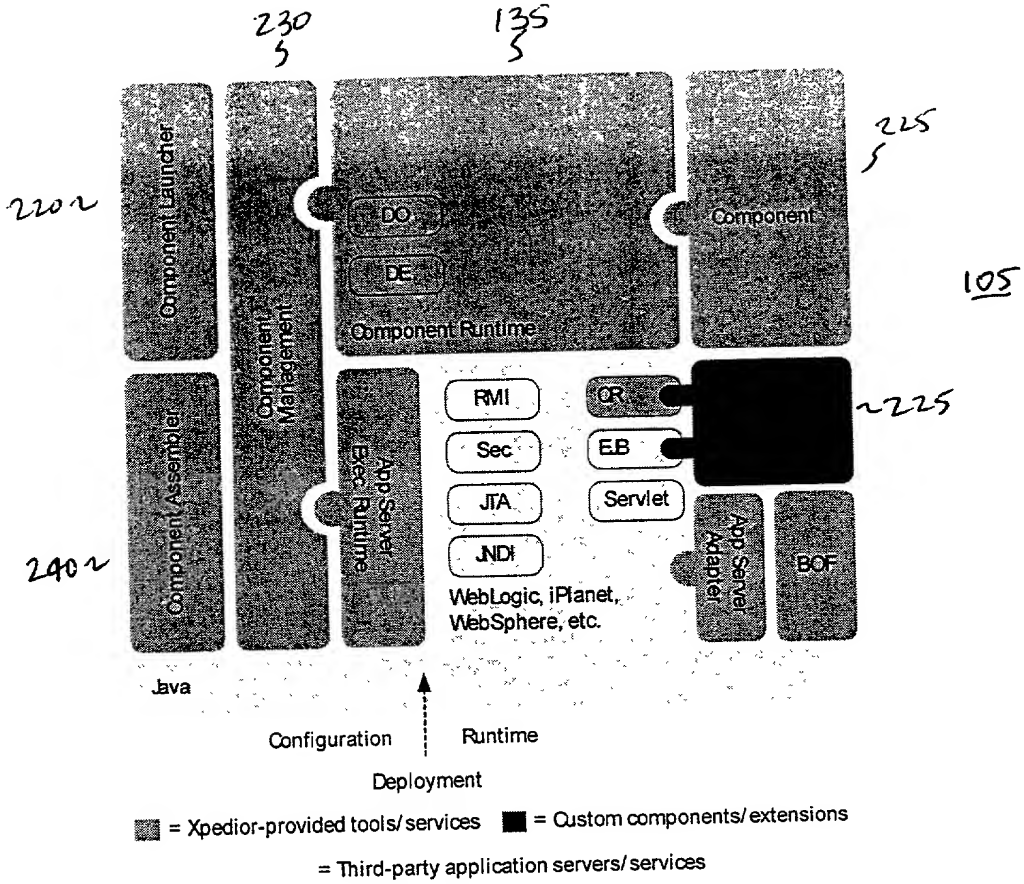


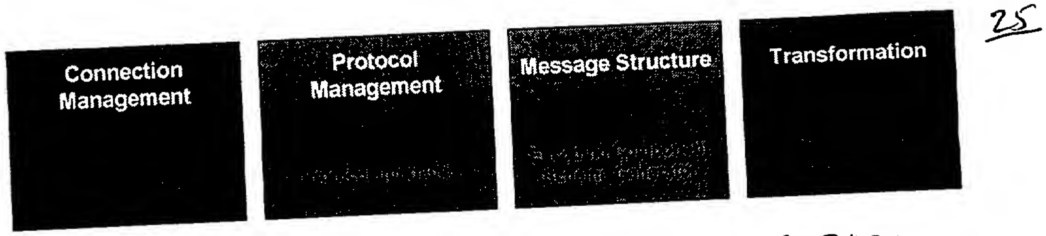
FIG. 7

~~*FIG. 1 is a diagram of elements and component layers of Enterprise Component Platform in the preferred embodiment.*~~



~~FIG. 4 is a block diagram of elements of a component platform according to the preferred embodiment.~~

FIG. 9



~~FIG 5 is a block diagram of elements of a messaging platform according to the preferred embodiment~~

FIG. 10

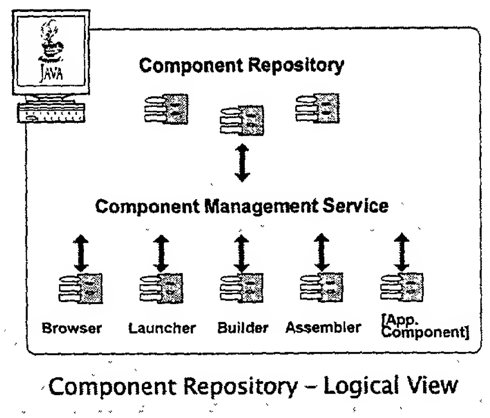


FIG. 11

~~FIG 6 The component repository~~

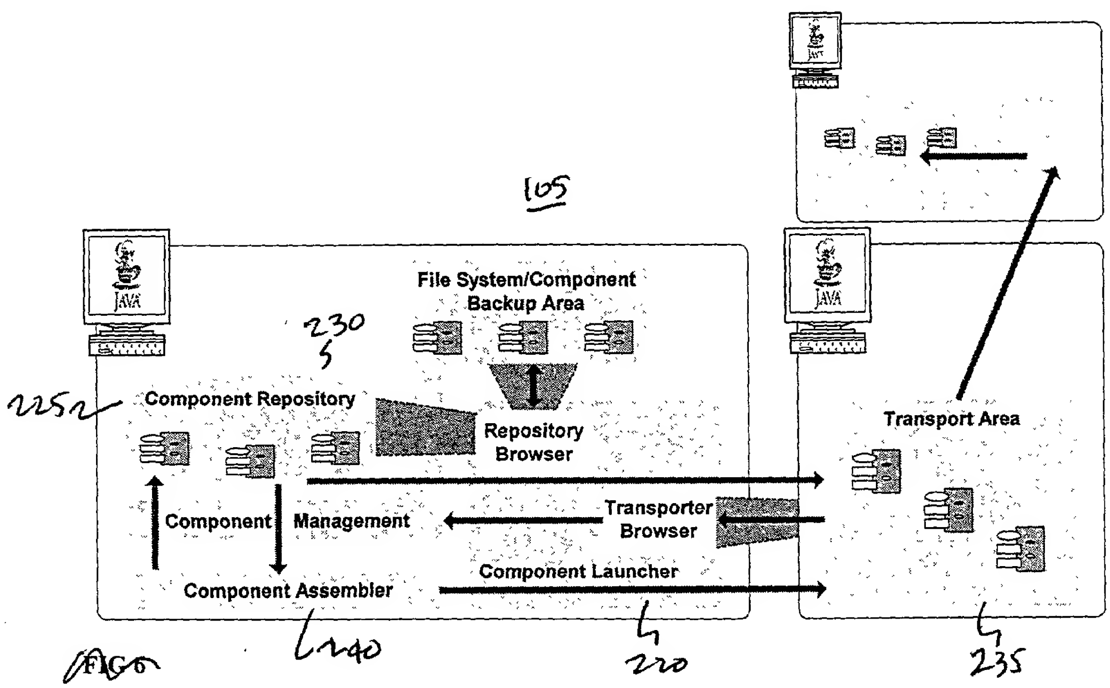


FIG. 12